UNIVERSITY OF OSLO
Department of Informatics

Verification of Network Simulators

The good, the bad and the ugly

Master thesis

Mats Rosbach

21st November 2012

Verification of Network Simulators

Mats Rosbach

21st November 2012

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisors, Andreas Petlund and Michael Welzl, for helping me with my thesis, answering question and providing valuable feedback. Thanks to the guys at the lab for keeping me company on our journeys to the coffee machine. And finally, thank you Kathinka, my dear, for kicking me out of bed in the morning... and for the moral support.

Abstract

The purpose of a simulation is to try to get an indication on how a given system will behave in a range of different scenarios. For this kind of research to have any value, it is important that the simulator models reality in a reliable way. In this thesis, we have chosen a common test suite through which we evaluate the TCP functionality of three major open-source simulators: ns-2, ns-3 and OMNeT++. As our set of test cases, we have used the TCP evaluation suite, which is meant to become a standardized test suite for evaluating TCP extensions. The suite already exists for ns-2, while we have implemented elements of the suite in both ns-3, and in OMNeT++. As a consequence of this, we have had to implement the tmix traffic generator as well for OMNeT++, and we have also worked on improving tmix for ns-3. These implementations of the test suite lead to some interesting observations regarding the TCP functionality of each network simulator. Through working with the test suite however, we have revealed several weaknesses in it. We have identified a range of items that need improvement before the test suite may become a useful tool.

Chapter 1

Introduction

1.1 Background and motivation

Network researchers must test Internet protocols under a variety of different network scenarios to determine whether they are robust and reliable. In some cases it is possible to build wide-area testbeds to perform such testing, but in many cases testing the protocol through a real world implementation is impossible. This is due to the complexity and difficulties in building, maintaining and configuring real world tests or testbeds. An alternative way of performing such testing is through the use of a network simulator.

The purpose of a simulation is to try to get an indication on how the protocol in question will behave in the real world. For this kind of research to have any value, it is important that the network simulator acts as it is supposed to do, i.e., like a real network. A network simulator, opposed to a real network, runs only as software. This means that every part of the network must be modelled and implemented by a developer. This, for instance, includes models of the physical equipment, such as cables or wireless signals, models of networking protocols, such as TCP and IP, or application-layer protocols, such as HTTP and FTP. Every detail from the real world cannot possibly be included in the simulation, and the problem is to decide which details are significant, and which are not. Choosing the correct abstraction level suitable for the simulation is one of the key challenges for the simulator developer. As an example of abstraction, consider the question on how time should be handled. Time as we know it in the real world is continuous, which means that there is no smallest amount of time that we can measure. If we measure nanoseconds for instance, there is always the possibility of measuring a smaller amount of time, namely picoseconds. In digital computers however, a smallest amount of time must be defined. When choosing an abstraction level, the scenario to be investigated must be considered. In network research we often measure delays in milliseconds or microseconds, so the simulator will obviously have to support measurements in microseconds. Hardware, such as central processing units (CPUs), works with much smaller amounts of time. The developer of a network simulator will at some point have

to make a decision and choose an appropriate abstraction level. This is a tradeoff between simulator performance, realism and simplicity.

Simulating real networks, and especially complex networks like the Internet, is difficult. Floyd and Paxson [21] concludes their discussion on simulating the Internet by saying that *"In many respects, simulating the Internet is fundamentally harder than simulation in other domains. In the Internet, due to scale, heterogeneity and dynamics, it can be difficult to evaluate the results from a single simulation or set of simulations. Researchers need to take great care in interpreting simulations results and drawing conclusions from them"*. This quote brings us to another potential problem with network simulators: actually using the simulator. A network simulator is a complex system that can be quite overwhelming to use. There are a lot of details that the researcher has to take into consideration while designing his simulation scenario. First of all there is the question of whether the scenario resembles a real world scenario. How realistic is the model? Then there is the problem of actually verifying that the model has been implemented as intended in the network simulator.

When publishing networking research papers, criticism is often based on the community's esteem of the simulator that was used. To the best of our knowledge, a systematic comparison of different simulators with reference to real life Internet tests has not been performed. In this work we aim to create test scenarios that can be used to compare and evaluate network simulators and analyse the results.

1.2 Problem definition

Network simulators are complex systems, and we have to limit this thesis to parts of the topic. We have chosen to focus our work on TCP and its congestion control algorithms. This means that we have chosen a set of test cases that specifically highlight the fundamental principles of TCP.

We have chosen a common set of test cases, which is implemented and evaluated in three open-source network simulators: ns-2, ns-3 and OMNeT++. We wanted to test commercial simulators as well, like OPNET [11] and EstiNet [3], but were unable to acquire them for our experimentation.[1]

As our set of test cases, we have chosen the TCP evaluation suite, which was originally the outcome of a round-table meeting, summarized by Floyd et al. [17], but has further been developed by David Hayes. The TCP evaluation suite describes a common set of test cases meant to be used by researchers to quickly get a basic evaluation of proposed TCP extensions. This evaluation suite seemed to have a reasonable set of test cases, and as a

[1] OPNET has a university program in which researchers and students may apply for a license of the simulator at discount prices. As stated on their web page however, one has to *"Agree that the software shall not be used for any comparative studies with competing simulation packages"* before applying for this license. This made it difficult for us to apply for such a license. We did not receive any reply from EstiNet when we contacted them about a license for our experiments.

side-effect, the implementation of the suite could be useful to the research community as well.

We have investigated three basic TCP variants: Tahoe, Reno, and NewReno. It would have been interesting to run newer variants as well (such as CUBIC), but due to limitations in the TCP functionality of ns-3 and OMNeT++, we have been unable to do so. The three variants are run through the TCP evaluation suite in each simulator to try to see whether each of them behaves in the way one would expect them to, but also to see how the three network simulators compare to each other. This thesis will investigate whether the differences between simulators are small, or whether the tested simulators deviate enough that the results may be questioned.

1.3 Contributions

There is no doubt that the network research community could benefit from having a standardized set of test cases, to be used as an initial evaluation of new TCP extensions. As part of this thesis, we have developed elements of the TCP evaluation suite in both OMNeT++ and ns-3 (the ns-2 version of the suite is already available online). As a consequence of this, we have also needed to implement the tmix traffic generator for OMNeT++, and have worked on improving tmix for ns-3.

Through the TCP evaluation suite implementation, we have evaluated the TCP functionality of the three network simulators. We have come to the conclusion that ns-2 has by far the richest TCP functionality of the three. ns-3 and OMNeT++ only have very basic TCP algorithms built-in: Tahoe, Reno and NewReno, as well as TCP without congestion control. In addition to this, both simulators are in the process of integrating the Network Simulation Cradle, which makes it possible to run real-world protocol stacks in a simulated environment. This would be a great enhancement to the TCP functionality of both of them.

The suite is still a work in progress, and will need to undergo major changes before it can be standardized and used as a tool to evaluate TCP extensions. We have evaluated the current state of the suite against three major open-source simulators. Through this evaluation, we have worked with the tmix traffic generator, which we argue has several flaws in its design that should be looked into. We have, as far as we know, been the first to try the TCP evaluation suite in practice, and our experience is that it is not at all ready yet.

1.4 Outline

The thesis is organized as follows. In chapter 2 we present background information relevant to our work. This includes general information about network simulators and emulators, TCP and congestion control, the TCP evaluation suite and tmix. In chapter 3 we present the evaluation suite

implementation in ns-2, ns-3, and OMNeT++ respectively. In chapter 4 we present our results, and we conclude the thesis in chapter 5.

Chapter 2

Background

This chapter contains background information necessary to understand this thesis. It will briefly summarize how systems can be evaluated, with focus on network simulators, it will draft the functionality of TCP and some of its variants. A common TCP evaluation suite has been proposed, and is described in this chapter, as well as a discussion on traffic generation in general.

2.1 Evaluation of systems

In general there are three different techniques to evaluate the performance of a system or a network; analytical modelling, simulation, or measurements [35]. All of these have their strengths and weaknesses. Each of the evaluation techniques is weak on their own, but a combination of them all should give thorough evaluation of the system.

Analytical modelling is a highly theoretical technique where a system is analysed by investigating mathematical models or formal proofs. The strength of such a technique is that it can be performed very early in the development process, before any time and money has been spent on actually implementing the system. It is a technique that evaluates the idea and the design of a system, and may give a basic indication on how well the system will perform, without going into specifics. The downside of analytical modelling is that it is largely based on simplifications and assumptions, as the actual system is not involved in the evaluation. This gives analytical modelling a low accuracy.

Simulations are closer to reality than analytical modelling, but still abstract away details. Simulations can be performed throughout the entire development process. They can early on give a good indication on how well the system will perform, but also be used in later stages of the process to find flaws in the design. Another strength of simulation, is that it is easy to perform large-scale tests that otherwise would be difficult to implement in a test-bed, or in the actual system itself.

Another technique of performing system evaluation is to take measurements of the actual system. Investigating the real system requires at least parts of the system be implemented. Testing on the real system will of

course give very accurate results, but at the cost of having to implement it before being able to evaluate it. Finding flaws in the design at this point is costly. The system does not have to be implemented fully, setting up a small-scale test-bed, or developing a prototype of the system is also possible, but this requires a lot of work as well.

ï»¿

2.2 Network simulators

2.2.1 Simulation models

A model is an object that strives to imitate something from the real world. This model cannot entirely recreate the real world object, and will have to abstract away some of its details. There are several types of simulation models out there. We will explain some of the major differences using a few basic attributes. Simulation models vary on whether they are dynamic or static. They may be deterministic or stochastic, and they may be discrete or continuous. These terms are important when describing models.

A dynamic model is a model that depends on time, in which state variables change over time. This could for instance be a model of a computer network that runs over several hours, where the state of each component varies depending on the traffic generated. A static model on the other hand, concerns itself with only snapshots of the system.

A deterministic model is a model in which the outcome can be determined by a set of input values. This means that if a simulation using a deterministic model is run several times using the same input values, the result will always be the same. In stochastic models on the other hand, randomness is present.

In discrete models, the state variables only change at a countable amount of times. Discrete models define the smallest unit of time supported, and events may only happen at these discrete times. This is opposed to continuous models, where state variables can change whenever, and the number of times in which events may happen is infinite.

2.2.2 What characterizes a network simulator?

A network simulator, as opposed to a real network, runs only as software. A network simulator must therefore consist of software that represents everything in a real network. This includes software for the physical equipment, like cables and wireless signals. It includes software that represents the connection points or the endpoints in the network, for instance a web server or a router. Depending on the network to be simulated, a protocol stack must be in place, for instance when simulating the Internet, network protocols such as IP, TCP and UDP must be in place. And lastly, some application (as well as a user using the application) is necessary to generate traffic on the network.

As already mentioned, simulations are based on different types of models. What attributes do we find in network simulators? It is possible

for network simulators to be either dynamic or static. A very common type of network simulator, a Discrete Event Simulator (DES), is a good example of dynamic simulator. These are simulators where events are scheduled dynamically as time passes. On the other hand, we have Monte Carlo simulations, which are static. This is a type of simulation that relies on repeated sampling to compute the result. Monte Carlo simulations are used to iteratively evaluate a model by using sets of random numbers as inputs. These types of simulations are often used when the model is complex and involves a lot of uncertain parameters. The model may be run many thousands of times with different random inputs, where the final result of the entire simulation typically is a statistical analysis of all the results combined. Monte Carlo simulation is especially useful for simulating phenomena with significant uncertainty in inputs, and for systems of high complexity. Computer networks however are usually well understood (at least compared to some mathematical and physical sciences where Monte Carlo simulations are usually used), and should not require an iterative method such as Monte Carlo to get representative results.

Computer simulations are in general discrete. The real world is continuous, as there are no rules that determine a smallest measure of time in which every event must occur. Mathematical models may be continuous. For instance, consider the function *f(x)*. The value of *f(x)* depends entirely on x, and as there are no limitations on the values of x, the range of values for *f(x)* is infinite. This is something that is difficult to achieve in computer simulations, and is only possible in analogue computers. When using digital computers, it is possible to achieve something close to a continuous simulation by making each time step really small, but it theory it will always be a discrete simulation. There is always the question of how much accuracy is really needed, and continuous time is something that network simulators abstract away from. For instance, in OMNeT++ it is possible to determine time down to a precision of 10^{-18} seconds. The question is: do we really need more?

As most network simulators are DESs, according to Wehrle et al. [35], we will only go into detail about DESs.

2.2.3 Discrete Event Simulators

DESs are described in [35], which includes most network simulators out there. A DES consists of a number of events that occur at discrete times during the run of the simulation. Discrete in this case means that the simulator is defined for a finite or countable set of values, i.e., not continuous, which means that there is defined a smallest measure of time in which an event can occur. Two consecutive events cannot occur closer together in time than the smallest measure of time dictates, although several events may occur at the same time. Below is a list of a few important terms in DESs:

- *An entity* is an object of interest in the system to be modelled. For instance, in a network simulation, an entity could be a node or a link

between two nodes.

- *A model* is an object in the simulator that imitates an entity from the real world. As an example we can use the model of a laptop. Most network simulators abstract away from most details of this laptop. In ns-3 for instance, all laptops, desktop computers, servers, routers, switches etc. in the network are simply called nodes. There is no information about the hardware of a node in ns-3 as a result of the abstraction level the designers chose when developing ns-3.

- *Attributes* describe certain aspects of an entity. A network channel may have attributes such as bandwidth and latency, while a traffic generator may have attributes such as inter-arrival time and average bit rate.

- *A system* consists of a set of entities and the relationships between them. This set of entities might include nodes, a wireless channel connecting the nodes, and models of a user that generates traffic through an HTTP application.

- *A discrete system* is a system where the states in the system only change at discrete points in time. The change of a state is triggered by some event in the system. One such change might be dequeuing a packet for transmission over a communication channel. This changes the state of the system as the packet is no longer located in the transmission queue, but is now located on the channel between nodes. Another event might be receiving a packet.

A simulation is a large model that is consisting of several sub-models that each represents real-world entities, a collection of abstractions in other words. In addition to the simulation model, a simulation also includes a way of generating input. The input is inserted into the simulator, which in turn makes the model act upon the input, thus generating output.

2.2.4 The use of network simulators

A network simulator imitates a specific environment as described in the previous section. This environment contains abstractions from the real world, abstractions for host machines, routers, wireless connections, fibre cables, user patterns and so on. Network simulators are typically used by researchers, engineers and developers to design various kinds of networks, simulate and then analyse various parameters on the network performance.

In network education, there is need for practical hands-on skills alongside the theory that students learn through a course. However, a dedicated network laboratory is expensive and is both difficult and expensive to maintain. A better alternative is to use a network simulator to give the students some kind of practical experience working with networks. Some network simulators, like ns-2, are designed for research and may be too complex and difficult to use for educational purposes [32].

Other network simulators are designed in a way that makes them more suitable for education. One example of such a simulator is Packet Tracer, which is an integral part of the Cisco Network Academy Program [25]. Network simulators used in commercial development is outside the scope of this thesis; however the benefits of using network simulators in research should also apply to development of commercial software.

2.3 Network emulators

An emulator is a piece of hardware or software that aims to duplicate (or emulate) the functions of a specific system. The emulator can be used as a replacement for systems. The focus of an emulator is to reproduce the *external* behaviour of the original system as closely as possible. This means that the emulator has to accept the same type of input that the original system does, and also that it has to produce the same output. In other words; the emulator has to perform the same *function* as the original system. Note that the emulator can implement this functionality however it wishes to, as long as it looks as if it functions in the same way that the original system does. Below are a couple of examples on how emulators might be used.

Emulation is a common strategy to help tackle obsolescence. The new versions of Windows are a good example of how this works. When a new version of Windows is shipped, there are functions from previous Windows versions that are left obsolete. The obvious problem is how older applications that rely on obsolete functions are supposed to continue working as intended. The most intuitive way of handling this is to actually update the application, but this is not always possible as many older applications are not maintained any more. Another way of handling this is to run the application on an emulator that emulates an older version of Windows. In Windows there is the possibility of running an application in *compatibility mode* [37], which emulates the older Windows version. To the applications, it seems like it is running on the older version of Window.

Another example is the Oracle VM VirtualBox [28], which emulates an entire computer. This can be used to run several operating systems on one machine, without having to actually install them on the real machine. It is useful for testing possibly malicious software, without running the risk of destroying anything useful on the computer. The operating system installed on the virtual machine will think that it is in fact installed on a real machine, because the emulator duplicates the functionality of a machine.

While the two previous examples were emulators, they were not network emulators. *Netem* is an example of a network emulator. Netem emulates certain properties of a wide area network. This includes having the possibility of modifying properties like delay, loss, duplication and re-ordering of packets. Netem appears to be an actual network to other systems. Netem implements send and receive functions that other systems may use in the same way that it might in a real network. Being able to "control" the network in this way is useful when testing and verifying new

applications and protocols.

2.4 Transmission Control Protocol

In the TCP/IP reference model [34], the transport layer is the second highest layer and communicates directly with the application on both the sending and receiving side. In general the basic function of the transport layer is to accept data from above, split it into smaller units if need be, and pass these units to the network layer. At the receiving end, the transport layer will accept data from the network layer and deliver them to the receiving application. In the classic OSI model, the transport layer only concerns itself with end-to-end communication, which means it is not involved in intermediate jumps on the route between sender and receiver. This is a simplification of the real world, as there are several protocols that break the layering of the OSI model. For instance, the Address Resolution Protocol (ARP) is a protocol that converts network layer addresses into link layer addresses, thus operating in both the link layer, and the network layer. Although the OSI model is highly simplified, it still gives a basic indication on what each type of protocol is supposed to do. The transport layer offers the application a set of services. The type of service offered however, depends on the transport layer protocol used. The three most commonly used transport layer protocols used in the Internet are: the Transmission Control Protocol (TCP) [31], the User Datagram Protocol (UDP) [30], and the Stream Control Transmission Protocol (SCTP) [33]. This section will describe the basics of TCP.

When the transport layer protocol hands over a segment to the network layer, there is no guarantee that the segment is ever delivered to the receiving end of the communication. TCP was specifically designed to provide a reliable end-to-end byte stream over an unreliable network (such as the Internet). The protocol offers simple send and receive functions to the application. These send/receive functions guarantee that all data are delivered in order and error-free. TCP is used in common application layer protocols such as HTTP (web), SSH (secure data communication) and SMTP (e-mail). The protocol has several features to provide its services in the best way possible:

- *Reliable end-to-end communication* - TCP ensures that every segment is delivered to the receiving endpoint. Before handing a segment to the network layer, a sequence number is assigned to the segment. When TCP receives a segment, it can use the sequence number to determine whether a segment has been lost, or whether a segment has been received out of order. The protocol replies with acknowledgements (ACKs) to tell the sender that a segment has been successfully received, and that the sender now can forget about this segment. By using sequence numbers in this way, the receiving end can detect segments that disappear on its route to the receiver. It will also detect whether any segments arrive out of order, and can reorder the segments before delivering them to the application.

Figure 2.1: Simple tcp example

Consider the simple example from Figure 2.1. In this example, we have a pair of communicating nodes, where the sender has sent six segments, numbered one through six. Segment number three has disappeared somewhere along the route towards the receiver. The normal technique to handle this is to ACK segment number two, which tells the sender that everything up to, and including, segment two has been received successfully. If no more ACKs are received, the sender must assume that everything after segment two is lost, therefore retransmitting segment three and upwards. This basic technique is known as *go-back-n*.

- *In-order delivery* - TCP also ensures that all segments are delivered in-order. This is quite easily done by buffering received segments that arrive out-of-order while waiting for the rest of the segments to fill in the gap. Once the missing segments have arrived, they will be delivered to the application. To the application it looks like all data arrived in-order.

- *Connection-orientation* - before communication can occur, a connection will have to be established between the two communicating nodes. The method for setting up a connection is known as the *three-way handshake*. When a node A wants to connect to a node B, it will first send a SYN (synchronize) packet to B. This SYN packet contains the initial sequence number of A. When B receives this packet, it will reply with a SYN-ACK. First of all it will ACK A's SYN packet, but it will also reply with its own SYN packet telling A the initial sequence number that B has chosen. When A receives B's SYN-ACK packet, it will ACK the SYN. When both SYNs have been ACKed, the connection is established.

  A TCP connection is identified through an addressing quadruple containing both the IP address and port number of both the endpoints. This connection establishment is necessary to keep track of sequence numbers and ACK numbers. A connection is maintained until one of the communication endpoints decides to tear down the connection, or until a given time without communication has occurred.

  Terminating a TCP connection is done as follows. When one of the communicating endpoints wants to terminate the connection, they send a segment with the FIN (finish) bit set, a so-called FIN segment. The receiver will ACK the FIN. Sending this FIN does not actually close the connection, but tells the receiver that the sender has stopped sending data. When the receiver is ready to close the connection, it will send its own FIN, which the sender will ACK. A connection is considered terminated when both sides have finished the shut

down procedure. A termination is not a three-way handshake like connection establishment, but rather a pair of two-way handshakes. Problems occur when packets are lost. For instance, consider the scenario where there are two nodes A and B. Node A has sent its FIN and received an ACK on its FIN. Eventually B wants to close its connection as well. B sends its FIN, but never receives an ACK from A. B does not know whether A received the FIN or not, and may retry sending the FIN. If A receives the FIN and replies with an ACK, how does A know whether B received the ACK or not? This problem has been proved to be unsolvable. A work-around in TCP is that both endpoints starts a timer when sending the FIN, and will close the connection when the timer timeouts if they have not heard from the other end.

- *Flow control* - TCP uses a flow control protocol to ensure that the sender does not overflow the receiver with more data than it can handle. This is necessary in scenarios where machines with diverse network speeds communicate, for instance when a PC sends data to a smartphone. In each TCP segment, the receiver specifies how much data it is willing to accept. The sender can only send up to the specified amount of data before it must wait for ACKs.

- *Congestion control* - a mechanism to avoid the network becoming congested. Congestion occurs when there is so much traffic on the network that it exceeds the capacity of the links in the network. This forces the routers to drop frames, which in turn will be retransmitted by TCP when the drop is eventually noticed. The retransmitted segments lead to more traffic, thus congesting the network even more, which again leads to more retransmissions. After a while, performance collapses completely and almost no segments are delivered. It is in everyone's interest that traffic is kept at such a level that congestion does not occur. TCP's congestion control mechanism lowers the sending rate whenever congestion is detected, as indicated by the missing ACKs.

2.4.1 Selective acknowledgements

A simple technique to handle retransmissions has already been mentioned, namely go-back-n. The obvious drawback of go-back-n is that every segment after a lost segment is retransmitted, regardless of whether they were successfully received or not. An improvement over this is the use of selective acknowledgements (SACK). Again, consider the simple example in Figure 2.1. In this example, segment three has been lost. Another way to handle the retransmission in this case, is to send a SACK back to the sender, with information on which segment has been dropped. Upon the reception of this SACK, the sender now knows that only segment three has been lost, and can therefore retransmit only the lost segment.

SACK is more complex than go-back-n, and leads to much more overhead when drop rate is high, but to retransmit only the dropped segments,

rather than retransmitting the dropped segment and the succeeding segments, can definitely be useful. Note that SACK is not a TCP variant on its own, but an alternative way of handling retransmissions that every TCP variant may use.

2.4.2 Nagle's algorithm

Nagle's algorithm [26] is an algorithm that improves the efficiency of TCP by reducing the number of packets that needs to be sent. The algorithm is named after John Nagle. He describes what he calls the "small packet problem". This is a scenario where TCP transmits a lot of small packets, for instance single characters originating from a keyboard. The problem with this is that for every single character that is to be sent, a full TCP/IP header is added on top. This means that for each byte the application wants to send, a 41 byte packet will be sent. This is obviously a problem in terms of efficiency. The algorithm is simple: If there is unconfirmed data in the pipe, buffer data while waiting for an acknowledgement. If the buffer reaches the size of a MSS, then send the segment right away. If there is no unconfirmed data in the pipe, send the data right away. This solves the small packet problem.

2.4.3 TCP Congestion control

Congestion control is one of the important features of TCP, as already mentioned briefly. This section will go through TCP's congestion control in more detail. There are several variants of TCP. Many of the earlier variants are no longer being used because newer and better algorithms have been developed, but there are still a number of variants out there. There is no answer to which algorithm is the best one, as each variant has strengths and weaknesses. Which variant to use depends on what attributes one is interested in. For instance, some variants focus entirely on getting the highest possible throughput, some variants focus on fairness between flows, and some variants focus on getting as low a delay as possible. We will describe some of the earlier variants first, and describe TCP congestion control as it has been improved over the years.

Tahoe

TCP Tahoe is one of the earliest TCP variants, and is quite basic compared to the variants that we have today. Tahoe uses three algorithms to perform congestion control: *slow start*, *congestion avoidance* and *fast retransmit*.

Both the slow start, and the congestion avoidance algorithms, must be used by a TCP sender to control the amount of unacknowledged data the sender can have in transit. The sender keeps track of this by implementing a congestion window that specifies how much outstanding data he can have at a given time, either in number of segments, or in number of bytes. If no congestion is detected in the network, the congestion window may

be increased. However, if congestion is detected, the congestion window must be decreased.

At the beginning of a transmission, the sender does not know the properties of the network. It will slowly start probing the network to determine the available capacity. Initially, the congestion window will be two to four times the sender's maximum segment size (MSS), which means that during the first transmission the sender can send two to four full-sized segments. When an ACK arrives, the sender may increase his congestion window by the size of one maximum segment. This means in theory that the congestion window doubles in size for every round-trip time (RTT) of the network. This exponential increase in window size continues until the slow start threshold, *ssthresh*, is reached. Initially, *ssthresh* should be set arbitrarily high so that the sender can quickly reach the limit of the network, rather than some arbitrary host limit by setting *ssthresh* too low. *ssthresh* is later adjusted when congestion occurs. Whenever the congestion window is smaller than *ssthresh*, the slow start algorithm is used, and whenever the congestion window is larger than *ssthresh*, the congestion avoidance algorithm is used.

During congestion avoidance, the congestion window is increased by roughly one full-sized segment per RTT. This linear increase in the congestion window continues until congestion has been detected. When congestion is detected, the sender must decrease *ssthresh* to half of the current congestion window before entering a new slow start phase, where the congestion window is set to the initial congestion window size.

The basic slow start and congestion avoidance algorithm is shown in Figure 2.2. In this figure we can see the congestion window starting at 1024 bytes at the first transmission (which means we have a maximum segment size of 512 bytes). For every transmission this congestion window doubles (increases exponentially) until it reaches the initial *ssthresh*, which is at 32KB. After *ssthresh* has been reached, the congestion window increases by 1024 bytes per transmission. After roughly 13 transmissions in this figure, a time-out occurs, which means that a segment probably is lost. This leads to *ssthresh* being lowered to half of the current congestion window, before starting a new slow start phase.

When a receiver receives an out-of-order segment, the receiver should immediately send a duplicate ACK message to the sender. From the receiver's point of view, an out-of-order segment can be caused by several network problems. The segment may arrive out-of-order because an earlier segment has been lost. It may arrive out-of-order because the order has been mixed up in the network, and finally a segment may arrive out-of-order because either a segment or an ACK has been replicated by the network. When a segment arrives out-of-order because an earlier segment has been lost, the chances are that several out-of-order segments will arrive at the same time. If for instance the sender has sent 10 segments at once, and segment number 4 has been lost, the receiver will receive the 3 first segments, then the last 6 segments roughly at the same time. This causes the receiver to send several duplicate ACK messages at the same time, one for each of 6 last segments. Therefore, upon the reception of 3 duplicate

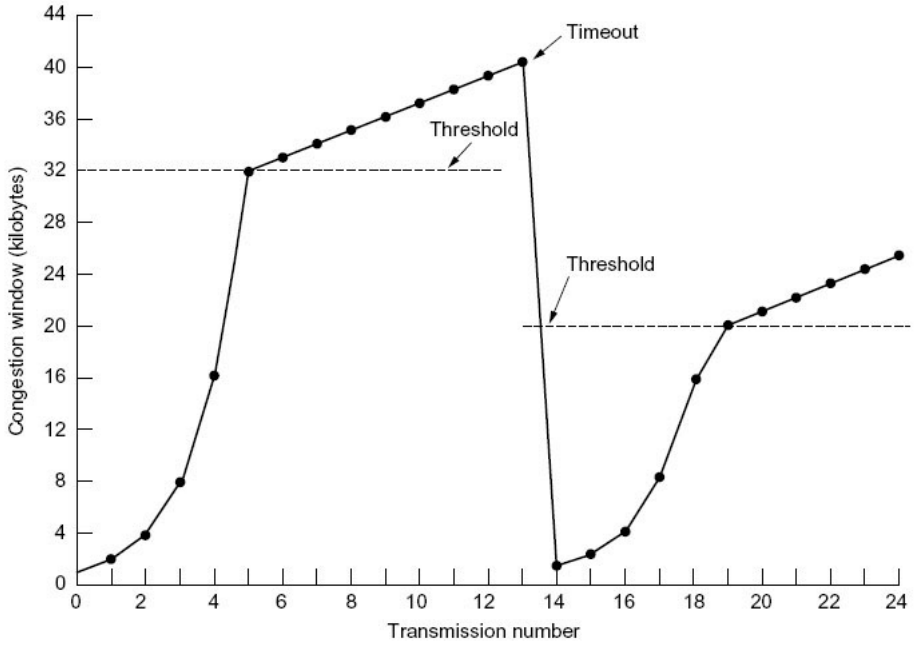


Figure 2.2: TCP's basic slow-start and congestion avoidance algorithm

ACK messages, the sender may conclude that a segment has very likely been lost, and thus resend the lost segment right away. This is called *fast retransmit*. Note that if two segments arrive out of order, for instance segment 4 and 5 have switches places along the traversal of the network, then only a single duplicate ACK will be sent back to the sender. As a single re-ordering is not a problem (the receiver will easily handle this himself), the sender should not retransmit the segment, and because the sender will not fast retransmit before he receives 3 duplicate ACKs, we can see that fast retransmit does work well with out-of-order segments as well.

Reno

TCP Tahoe includes three basic algorithms: slow start, congestion avoidance and fast retransmit. TCP Reno, described in [15], introduces a new algorithm to improve the fast retransmission of a packet: *fast recovery*. TCP Reno is out-dated today, but many of the newer TCP variants are based on TCP Reno.

In the case of a fast retransmit, the sender must lower *ssthresh* as it normally would when a segment is lost. Instead of going into a slow start phase however, the sender should set its congestion window to the new *ssthresh*, and go straight into congestion avoidance. This is because of the fact that receiving duplicate ACKs not only indicates that a segment has been lost, but it also indicates that some of the succeeding segments

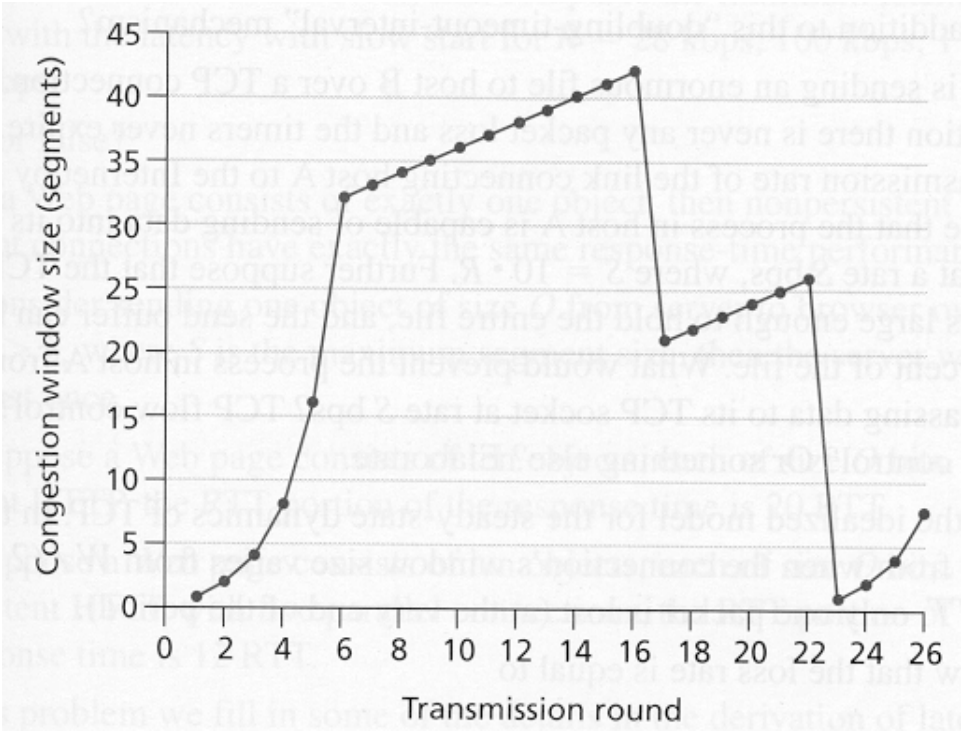


Figure 2.3: TCP Reno with fast retransmit and fast recovery

were successfully received by the receiver. This means that the succeeding segments are no longer consuming network resources, and therefore the sender may conclude that the network is not fully congested, and that a single segment has been dropped by chance. This is called *fast recovery*.

TCP Reno is shown in Figure 2.3. We can see that same slow start and congestion avoidance phases that we saw in TCP Tahoe. After roughly 16 transmission rounds, a fast retransmission occurs (due to three duplicate ACKs). *ssthresh* is lowered to half of the current congestion window, but it does not enter a new slow start phase, it goes straight into congestion avoidance. After roughly 22 transmission rounds however, a time-out occurs, thus triggering a new slow start phase.

2.4.4 NewReno

TCP NewReno is an improvement of the Reno variant, and is described in detail in rfc3782[20]. Segment retransmissions only occur whenever a retransmit timer timed out, or when three duplicate ACKs have arrived. In the latter case the fast retransmit algorithm is triggered. The problem with the fast retransmit algorithm is that the sender assumes that only a single segment was lost. The sender does receive several ACKs, hinting that at least some of the sent segments were received, but there is no way the sender can know which packets were successfully delivered, and which packets were dropped, unless SACK is enabled. The only thing that the sender knows for certain is that the segment indicated by the duplicate

ACKs is lost. When several segments from the same window are lost, Reno is not very effective, because it will take a full RTT before the retransmitted segment is ACKed, and therefore it will also take a full RTT for each lost segment to discover new lost segments. Another problem is that *ssthresh* is halved each time Reno fast retransmits a segment, which means *ssthresh* is halved for each segment lost, as it treats each loss individually. Note that NewReno only applies to connections that are unable to use the SACK option. When SACK is enabled, the sender has enough information about the situation to handle it intelligently.

TCP NewReno differs from Reno on how it implements the fast retransmit and fast recovery algorithms. Upon entering fast recovery (after receiving 3 duplicate ACKs), *ssthresh* will be lowered to half of the current congestion window (cwnd), and then *cwnd* is set to *ssthresh + 3 * SMSS*. Reno will leave this fast recovery phase as soon as a non duplicate ACK is received, NewReno however does not. NewReno stays in the same fast recovery phase until every segment in the window has been ACKed, thus lowering *ssthresh* only once.

NewReno works with what is called a *partial acknowledgement*, which is an ACK that does not ACK the entire window. As long as the sender receives partial ACKs, it will not leave the fast retransmit phase, but NewReno also includes another improvement over Reno. For every duplicate ACK the sender receives, it will send a new unsent segment from the end of the congestion window. This is because the sender assumes that every duplicate ACK stems from a successfully delivered segment, and therefore these segments are no longer in transmit, thus opening up for new segments to be sent.

We can see that NewReno, just like Reno, still has to wait a full RTT to discover each lost segment. However, not halving *ssthresh* for each segment lost, and adding new segments to the transmit window for every duplicate ACK received, is still a great improvement over Reno.

2.4.5 BIC and CUBIC

TCP BIC and TCP CUBIC [2] are two variants of TCP that focus on utilizing high-bandwidth networks with high latency. As demands for fast downloads of large-sized data are increasing, so is the demand for a TCP variant that can actually handle these transfers efficiently. The problem is that over high-latency networks, TCP does not react quickly enough, thus leaving bandwidth unused. TCP BIC is a congestion control protocol designed to address this problem.

BIC has a very unique window growth function which is quite different to the normal TCP slow start and congestion avoidance algorithms. What is interesting is what happens when a segment is lost. When BIC notices a loss, it reduces its window by a multiplicative factor. The window size just before the reduction is set to maximum (Wmax), while the window size after the reduction is set to minimum (Wmin). The idea is that because a segment was lost at Wmax, equilibrium for this connection should be somewhere between Wmin and Wmax. BIC performs a binary search

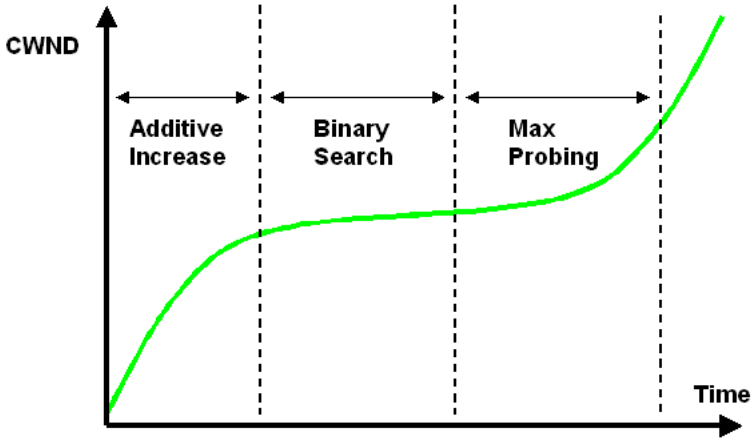


Figure 2.4: BIC and its three phases

between Wmin and Wmax to quickly try to find the equilibrium. For each RTT after the segment loss, BIC jumps to the midpoint between Wmin and Wmax, thus increasing rather quickly. If a new segment loss does not occur, the new window size also becomes the new Wmin. However, there is a max limit to how fast it may increase each RTT; the limit is a fixed constant, named Smax. If the distance between the midpoint and Wmin is larger than Smax, then BIC will increase by Smax instead. This means that often after a segment loss, BIC will first increase additively by Smax each RTT, and then when the distance between Wmax and Wmin is smaller, it will use a binary search, thus increasing logarithmically. We can see that BIC increases very rapidly after a packet loss for a while, and then it slows down as it closes in on what it assumes to be the maximum window size that the connection can handle. This binary search continues until the congestion window increase is smaller than some fixed constant Smin. When this limit is reached, the current window size is set to the current maximum, thus ending the binary search phase. When BIC reaches Wmax, it assumes that something must have changed in the network, and that the current maximum most likely is larger than Wmax. It then goes into a max probing phase where it tries to find a new maximum. The growth function during this phase is the inverse of those in additive increase and binary search. First it grows exponentially. Then after a while, after reaching a fixed constant, it does an additive increase. This means that BIC is looking for a new Wmax close to the old maximum first, but when it cannot find it, it will quickly look for a new one further away from the old maximum. BIC's growth function is summarized in Figure 2.4.

Although BIC achieves pretty good scalability, fairness and stability, BIC can be a bit too aggressive compared to other variants, especially when the RTT is low [2]. This is because BIC is focusing on utilizing the link capacity, and always tries to stay close to the maximum. BIC is also quite a complex TCP variant. CUBIC simplifies the window control, as well as enhances BIC's TCP friendliness.

Instead of having several phases and fixed constants, CUBIC applies

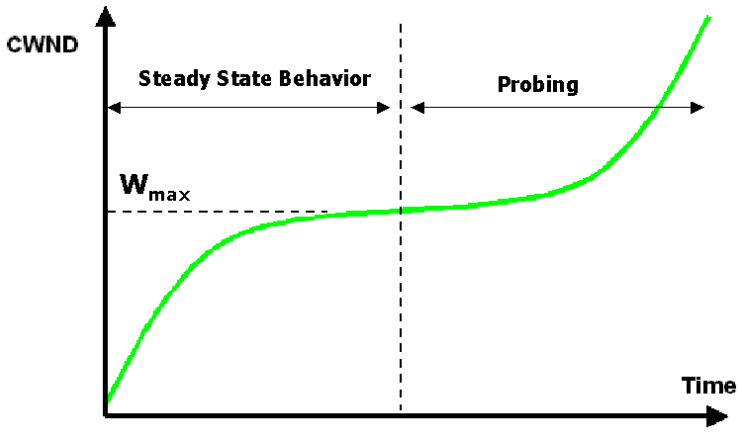


Figure 2.5: CUBIC's growth function

a single cubic function to determine the size of the congestion window. Figure 2.5 shows CUBIC's growth function. This growth function is generated from a simple cubic function, and we can see that it is very similar to BIC's growth function. The cubic function that CUBIC uses grows slower than BIC's growth function, thus ensuring a better TCP friendliness. Applying only a single function also gives a much simpler algorithm. All in all, we could say that CUBIC is a simplified version of BIC.

2.5 Queue management

While TCP contains a number of congestion control algorithms that have their strengths and weaknesses, there are also other management systems that helps out with congestion control. How well congestion is handled in a network not only depends on TCP, but it also depends on how the link-layer handles congestion. Note that link-layer congestion control is not a part of TCP, but is necessary to understand in order to understand congestion as a whole. This section will describe a few basic queuing disciplines.

The most basic queue is the drop-tail queue. This is a very simple queue management algorithm. All the traffic is treated the same, no traffic is differentiated. With tail drop, the queue accepts packets until it reaches its maximum capacity. When this capacity is met, the newly arriving packets are dropped. This is a very simple and intuitive scheme.

There are more complex queuing algorithms with more functionality. This is known as Active Queue management (AQM). According to Tanenbaum [34]: *"It is well known that dealing with congestion after it is first detected is more effective than letting it gum up the works and then trying to deal with it"*. This is the basic idea behind AQM. What if packets are dropped or marked before the network is actually congested?

A popular queuing scheme is called Random Early Detection (RED). The idea behind this queue is to randomly select a few packets and drop

them when the queue is nearing its capacity. The point of this is to activate the sender's congestion control algorithm (given that the sender implements such a thing) before the network is congested. Note that the queue does not know anything about which source is causing most of the trouble, so dropping packets at random is probably as good as it can do.

There are a lot of other queuing disciplines out there that function quite differently from drop-tail queues and RED queues. Fair queuing for instance, which tries to let each flow in the network have their piece of the capacity. These queuing disciplines however are outside the scope of this thesis.

2.6 Traffic generation

When simulating a network, we also need traffic to populate our network. In general, traffic generation can take either an open-loop approach, or a closed-loop approach. An open-loop system is by far the simpler of the two. In an open-loop system there is some sort of input (for instance a mathematical formula to produce a stream of packets), which is injected into the network, and produces some kind of output. An open-loop system is very simple in the fact that it does not adjust itself based on feedback from the network, but just "blindly" injects traffic into it. A closed-loop approach to traffic generation on the other hand, generates traffic, monitors the network, and decides how to further generate more traffic. Michele Weigle et al. [36] states that *"The open-loop approach is now largely deprecated as it ignores the essential role of congestion control in shaping packet-level traffic arrival processes"* What this means is that generating traffic blindly in an open-loop manner, without adapting to the response that the network offers, is very unrealistic. Also note that a congested network, i.e., a slow and sometimes unresponsive network, will actually alter the way the user interacts with an application. For instance a user might give up on a web page if it takes too long to get a hold of it. Floyd et al. [21] states that *"Traffic generation is one of the key challenges in modelling and simulating the internet"*. One way of generating traffic is to base the generator on traffic captured on a real network, i.e., a trace-driven simulation. Trace-driven simulation might appear to be an obvious way to generate realistic traffic as seen in the real world. The problem however, is that the network on which the trace was captured most likely uses a different topology, a different type of cross-traffic, and in general is quite different from the simulated network on which the generator is being used. The timing of a single packet from a connection only reflects the conditions of the network at the time the connection occurred. Due to the different condition of the simulated network, a trace-driven simulation will still have to react on its own to the network. This means that the traces will have to be *shaped*. The problem with shaping a trace like this is that if you reuse a connection's packets in another context, the connection behaves differently in the new context than it did in the original. This does not mean that shaping traffic based off of traces is pointless. It is still possible to deduce some kind of source

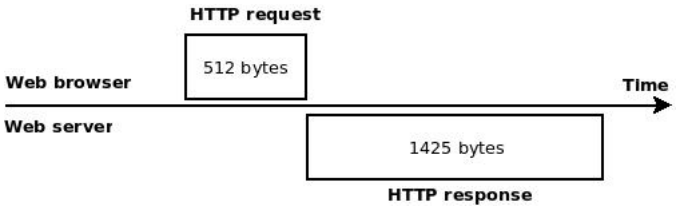


Figure 2.6: Typical HTTP session

behaviour that reassembles plausible traffic, although not perfect. Another difficulty in generating traffic is to choose to what level the traffic should congest the network links. "*We find wide ranges of behaviour, such that we must exercise great caution in regarding any aspect of packet dynamics as 'typical'*" [29]. This leads to the conclusion that whenever modelling and simulating a network, it is very important not to focus on a particular scenario at a particular level of congestion, but rather explore the scenario over a range of different congestion levels.

2.6.1 Tmix - A tool to generate trace-driven traffic

In this thesis, where test scenarios are based off of the TCP evaluation suite described in Section 2.7, we have a used a tool named tmix to generate our traffic. This is the same traffic generator used in the ns-2 version of the suite. In ns-2, this traffic generator is part of the standard codebase as of version ns-2.34. In ns-3, tmix comes as a standalone module, which we have modified slightly for our purpose. Unfortunately there is nothing close to an application like tmix in OMNeT++; therefore we have implemented the core functionality of tmix for OMNeT++. This section contains a short summary of Tmix as described in [36] and [27].

Tmix tries to recreate traffic as seen on an actual network. The approach taken is to not focus on traffic generated by single applications, but rather focus on traffic as a whole on the network, independent of applications. It is based on the fact that most application-level protocols are based on a few simple patterns of data exchanges. Take the HTTP protocol for instance. HTTP is based on a very simple client-server architecture, where the server has information (a web page for instance) that the client is interested in. This results in the client sending a single request to the server, and the server responding with the requested object. More generally, a client makes a series of requests to the server, and after each request the server will respond. Before making a new request, the client will wait for an answer. This is known as a sequential connection and is typically seen in applications such as HTTP, FTP-CONTROL and SMTP. An example of a typical HTTP session is shown in Figure 2.6.

Another pattern of traffic often seen on the Internet is a form of traffic where the application data units (ADUs) (which is a packet of data that the application passes to the transport layer) from both endpoints in a connection are more or less independent of each other. This is known as a concurrent connection and is typically seen in application-layer protocols

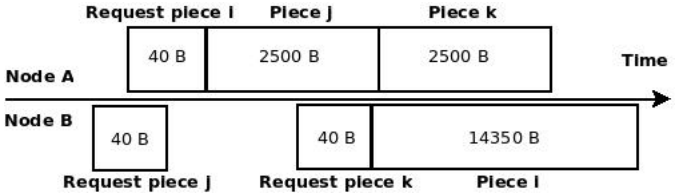


Figure 2.7: Simplified BitTorrent session

such as BitTorrent. An example of a simplified BitTorrent session is shown in Figure 2.7, where data flows in both direction seemingly independent of each other.

Tmix traffic occurs between two nodes in a network, the so-called initiator and acceptor. They transfer ADUs between each other. A series of ADU exchanges is represented as a connection vector that describes each and every ADU sent between the initiator-acceptor pair. Each connection vector specifies a number of common attributes for the entire series of exchanges, such as maximum segment size (MSS) and minimum round-trip time (RTT). After the common attributes, each connection vector specifies a number of epochs of the form *(a-b-t)*, where *a* is the size of the ADU sent from the initiator to the acceptor, *b*, is the size of the ADU sent from the acceptor to the initiator, and *t* is the think time between receiving the response and sending the next request. Note that it is possible to omit an ADU during an epoch, by having either a or b be zero. Also note that [27] specifies two different types of connection vector formats. We will only describe the Alternate Connection Vector Format as that is the format we are using. Below is an example on how both concurrent and sequential connection vectors might look like:

```
S 6851 1 21271 55538
w 64800 6430
r 9877
l 0.000000 0.000000
I 0 0 245
A 0 51227 510
A 6304943 0 0

C 1429381 2 2 21217 55382
w 64800 6432
r 10988
l 0.000000 0.000000
I 0 0 222
A 0 0 726
I 1242237 0 16
A 1444888 0 102
I 726 0 0
A 102 0 0
```

The first connection vector starts with a '**S**'. This means we have a sequen-

tial connection. The rest of the first line is the start time (in microseconds), the number of epochs in the connection, and two identification numbers. In this case we have a sequential connection with one epoch, starting at 6,851 ms (where 0 is the start of the simulation). The second line in a connection vector, starting with a '**w**', gives the window sizes of the initiator and acceptor, respectively. The third line gives us a minimum RTT (in microseconds), while the fourth line, which is the last line before the actual ADUs, tells us the drop rate of the connection. For the actual ADUs, we have lines starting with '**I**', which means an ADU sent from the initiator, and lines starting with '**A**' which means an ADU sent from the acceptor. The second and third field of these lines are two different types of wait times. Note that one of these has to be zero. If the first one is non-zero, it means that we should wait an amount of time after *sending* our previous ADU, before we send the next ADU. If the second one is non-zero, it means that we should wait an amount of time after *receiving* an ADU, before we send the next one. In our first connection we first have an ADU of size 245 sent from the initiator at time 0 (after the connection vector start). Upon reception of this ADU, the acceptor waits for 51,227 ms before replying with 510 bytes, and then waits 6,3 seconds before closing the connection. We see that the last ADU is of size zero, this is a special placeholder ADU that means we should close the connection. The second connection vector is quite similar to the first one. It starts with a '**C**', which tells us we have a concurrent connection. It specifies two numbers of epochs, as opposed to the one that the sequential connection specified. This is needed as the initiator and acceptor are independent of each other, and may have a different amount of epochs. The rest of the attributes preceding the ADUs, are the same for concurrent connections as for sequential connections. The actual ADUs are self-explanatory. Note that the third field of each ADU is never used, because the endpoints in a concurrent connection never wait on ADUs from its counterpoint. Also note that both ends close their connection.

2.7 The TCP evaluation suite

There is no doubt that the network research community could benefit from having a standardized set of test cases, to be used as an initial evaluation of new TCP extensions. Such an evaluation suite would allow researchers proposing new TCP extensions to quickly and easily evaluate their proposed extensions through a series of well-defined, standard test cases, and also to compare newly proposed extensions with other proposals and standard TCP versions. A common TCP evaluation suite has been proposed by [17] and has been further developed by David Hayes [22] [1].

[1]The TCP evaluation suite is still a work in progress, meaning that some of the suite will change, and some of the values have yet to be decided. The version cited is the latest version *released*. There exists newer version. We have based our work on a snapshot of the suite that we have acquired through private correspondence with David Hayes. There are a few differences between the released version and the one we have. The differences will

In this chapter we will give a brief summary of the TCP evaluation suite as described in the draft we received from Hayes.

2.7.1 Basic scenarios

There are a number of basic scenarios described in the evaluation suite: The data center, the access link, the trans-oceanic link, the geostationary satellite, the wireless access scenario, and the dial-up link. All of these scenarios are meant to model different networks as seen in the real world.

The access link scenario for instance, models a network connecting an institution, such as the University of Oslo, to an ISP. The central link (CL) itself has a very high bandwidth, and low delay. The leaf nodes on the institution represent a group of users with varying amount of delay connected to the institution's network. Some of these users are perhaps on campus, and are connected to the network through wireless access, thus giving a high delay, while some of the users are directly connected to the network by high-speed cabling, thus giving short delays.

The trans-oceanic link on the other hand, models the high capacity links connecting countries and continents to each other. The CL has a really high bandwidth, but also a high propagation delay because of the long distances. The leaf nodes represent different types of users, with a varying amount of delay, connected to the trans-oceanic link.

There are not only high-speed scenarios defined in the suite, there is also the dial-up link scenario for instance. This models a network where several users are connected to the same 64 kbps dial-up link. Scenarios like these are reported as typical in Africa.

Each scenario is run for a set amount of time. The simulation run time is different for each scenario. Some scenarios, like the access link scenarios, have really high traffic and does not require as much time to achieve reasonable results, while others, like the dial-up link scenario, requires more time to achieve reasonable results. Every scenario also has a warm-up period, in which measurements are not taken. This is to let the network stabilize before taking measurements. The actual numbers are just placeholder values in the version of the suite we have used, and are therefore not described here. The values we have used in this thesis can be seen in the chapter about the ns-2 version of the suite.

Topology, node and link configurations

All of the basic scenarios use the same topology: a simple dumbbell topology consisting of two routers connected to each other through the CL. Each of the two routers is connected to three leaf nodes, where each node represents a group of users. The six leaf nodes are numbered one through six. The topology is pictured in Figure 2.8.

There is a standard set of propagation delays for the edge links that is used in most of the basic scenarios. Respectively, these link delays are

be discussed in the description of our implementation.

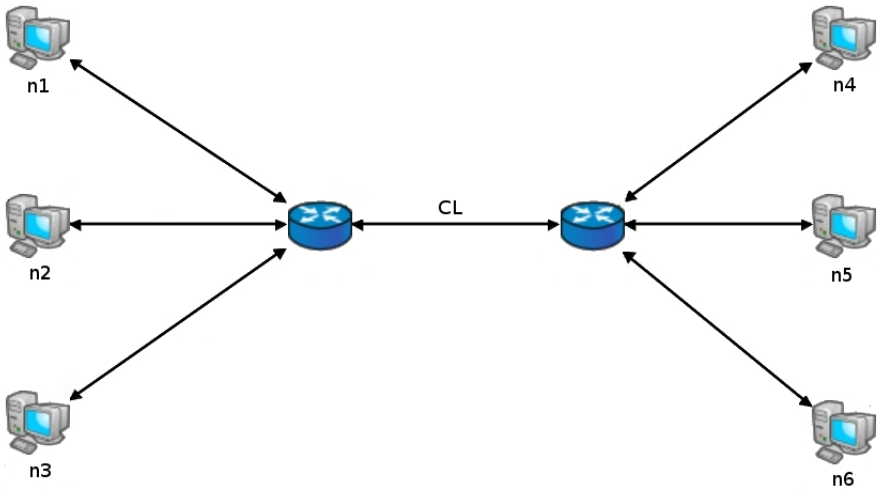


Figure 2.8: Simple dumbbell topology with standard delays

Path	RTT	Path	RTT	Path	RTT
1-4	4	1-5	74	1-6	150
2-4	28	2-5	98	2-6	174
3-4	54	3-5	124	3-6	200

Table 2.1: The distribution of RTTs for standard propagation delays

0ms, 12ms, 25ms, 2ms, 37ms, and 75ms. These delays are defined in such a way that every path between a pair of nodes will have a different round-trip time (RTT). Table 2.1 shows the RTT for each path in the dumbbell topology.

The set of delays for each basic scenario is shown in Table 2.2. Delay is measured in milliseconds, and CL denotes the delay of the CL, while n1, n2, ..., n6 shows the delay of the links connecting node n1, n2, ..., n6 to the CL. In the case of the wireless access scenario, n4, n5 and n6 shows the one way propagation delay of the link connecting them to their corresponding wireless access point, which in turn is connected to the central link. The bandwidths (measured in Mbps) for each scenario is shown in Table 2.3, where CL L->R is the central link from the left side to the right side, and CL R->L is the central link in the opposite direction. n1, n2, ..., n6 shows the bandwidth of the link connecting node n1, n2, ..., n6 to the central link.

Queuing has a great impact on how TCP behaves. The TCP evaluation suite does not decide what type of queue to use for the basic scenarios, so this is up to the user. There is one scenario however, summarized later, which aims to show how different queuing disciplines affect the different TCP variants. Each scenario has different queue lengths, measured in milliseconds. A queue length of 100ms for instance, means a queue where a single byte should never have to wait more than 100ms before being transmitted onto the link. Table 2.4 shows the different queue sizes along with the maximum number of packets each queue should be able to hold

Scenario	CL	n1	n2	n3	n4	n5	n6
Data center	0	1	1	10	1	10	10
Access link	2	0	12	25	2	37	75
Trans-oceanic link	65	0	12	25	2	37	75
Geostationary satellite	300	0	12	25	2	37	75
Wireless access	2	0	12	25	2	37	75
Dial-up link	5	0	12	25	2	37	75

Table 2.2: Delays for each basic scenario, in milliseconds

Scenario	CL L->R	CL R->L	n1	n2	n3	n4	n5	n6
Data center	1000	1000	1000	1000	1000	1000	1000	1000
Access link	100	100	100	100	100	100	100	100
Trans-oceanic link	1000	1000	1000	1000	1000	1000	1000	1000
Geostationary satellite	40	4	100	100	100	100	100	100
Wireless access	100	100	N/A	N/A	N/A	N/A	N/A	N/A
Dial-up link	0.064	0.064	N/A	N/A	N/A	N/A	N/A	N/A

Table 2.3: Bandwidth for each basic scenario, in Mbps

for 1500B packets. To calculate the buffer size in number of packets:

$$\textit{buffer size(packets)} = \frac{\textit{link capacity} * \textit{buffer size(seconds)}}{\textit{maximum packet size}}$$

As an example, for the access link scenario with a maximum packet size of 1500 bytes, we have:

$$\textit{buffer size(packets)} = \frac{1Mbps/8 * 0.1s}{1500B} = 833 packets$$

Traffic

For this test suite, network traffic consists of sessions corresponding to individual users. As each user is independent, session arrivals are well modelled by an open-loop Poisson process. A user session may consist

Scenario	size in ms		size in packets	
	router1	router2	router1	router2
Data center	10	10	833	833
Access link	100	100	833	833
Trans-oceanic link	100	100	8333	8333
Geostationary satellite	100	1000	333	333
Wireless access	N/A	N/A	N/A	N/A
Dial-up link	2000	2000	11	11

Table 2.4: Buffer sizes for each scenario

Scenario	No congestion	Mild congestion	Moderate congestion
Data center	TBD	0.5%	1.5%
Access link	TBD	0.5%	1.5%
Trans-oceanic link	TBD	0.5%	1.5%
Geostationary satellite	TBD	3%	6%
Wireless access	TBD	1%	3%
Dial-up link	TBD	5%	15%

Table 2.5: The drop rate for each level of congestion

of a single greedy TCP flow, several greedy TCP flows separated by user think times, or a single non-greedy flow with embedded think times. As described in Section 2.6.1, we use tmix to generate such traffic.

Each scenario is run three times with three levels of congestion: no congestion, mild congestion and moderate congestion. Congestion in this case is measured in average drop rate. The actual numbers for each scenario varies, and is summarized in Table 2.5. Some of these values have not been finalized yet, the actual numbers when running with no congestion has yet to be decided.

Output

For the CL in both directions the following metrics are to be collected:

- The aggregate link utilization (in percentage)
- The average packet drop rate (in percentage)
- The average queuing delay (in milliseconds)

By "link utilization" we mean the percentage of the link's bandwidth that is currently being consumed by network traffic. By traffic in this case we mean all data bits that are sent on the link, including both payload as well as packet headers. By "aggregate link utilization" we mean the average link utilization throughout the whole run of the test case (except for the warm-up time). This gives us the following formula for calculating the aggregate link utilization:

$$\text{Aggregate link utilization} = \frac{throughput * 100}{\text{link capacity}}$$

The average packet drop rate is quite self-explanatory. It is the ratio of dropped packets on the central link, compared to the number of packets that were sent to the link. This gives us the following formula:

$$\text{Packet drop rate} = \frac{\text{lost packets}}{\text{successfully transmitted packets + lost packets}}$$

The average queuing delay is the amount of time a traversing packet spends waiting in queue at the two central link routers. This is the difference in time between a packet enqueuing and dequeuing at both the

routers. As the link capacities of the edge links are always larger than the capacity of the central link, we can assume that the queuing delay for the edge devices are 0, or close to 0. This gives us the following formula:

$$\text{Queuing delay} = (T2 - T1) + (T4 - T3)$$

where T2 is the timestamp on dequeuing from the first router, T1 is the timestamp on enqueuing in the first router. T4 is the timestamp on dequeuing from the second router, and T3 is the timestamp on enqueuing in the second router. In addition to metrics collected for the central link, there are a number of metrics for each *flow*:

- The sending rate (in bps)
- Good-put (in bps)
- Cumulative loss (in packets)
- Queuing delay (in milliseconds)

The *sending rate* of a flow is the number of bits transmitted per second. This includes both header and data, from both successfully transmitted packets, as well as packets that are eventually dropped. *Good-put* on the other hand, only measures the throughput of the original data. This means each and every application level bit of data that is successfully transferred to the receiver, excluding retransmitted packets and packet overhead. *Cumulative loss* is the number of lost packets or the number of lost bytes. *Queuing delay* is also measured for each flow, and is similar to the queuing delay described for the central link. It is measured for each flow individually, and in addition to the central link queues, there may be other queues along the network which has to be taken into consideration.

2.7.2 The delay-throughput tradeoff scenario

Different queue management mechanisms have different delay-throughput tradeoffs. For instance, Adaptive Virtual Queue (AQM) gives low delay, at the expense of lower throughput. The delay-throughput tradeoff (DTT) scenario investigates how TCP variants behave in the presence of different queue management mechanisms.

The tests in this scenario are run for the access link scenario. It is only run for one level of congestion: moderate congestion when the queue size is 100% of the bandwidth-delay product (BDP). The scenario is run five times for each queuing mechanism. When using drop-tail queues, the scenario is run five times with queue sizes of 10%, 20%, 50%, 100%, and 200% of the BDP. For each AQM scenario (if used), five tests are run, with queue sizes of 2.5%, 5%, 10%, 20%, and 50% of the BDP.

For each scenario, the output should be two graphs. One shows the average throughput as a function of average queuing delay, while the other graph shows the packet drop rate as a function of average queuing delay.

2.7.3 Other scenarios

We have described the most basic scenarios in detail in the preceding sections, and will briefly mention the rest of the scenarios in this section, so that the reader can get an impression of the range of tests that is included in the suite. There is one test named *"Ramp up time"*. This test aims to determine how quickly existing flows make room for new flows. The *"Transients"* scenario investigates the impact of sudden changes in congestion. This scenario observes what happens to single long-lived TCP flow as congestion suddenly appears or disappears. One scenario aims to show how the tested TCP extension impacts standard TCP traffic (where the standard TCP variant is defined to be NewReno). This scenario is run twice: once with only the standard TCP variant, and once where half the traffic is generated by a standard TCP variant, and the other half is generated by the TCP extension to be evaluated. The results of these two runs are compared. The *"Intra-protocol and inter-RTT fairness"* scenario aims to measure bottleneck bandwidth sharing among flows using the same protocol going through the same routing path, but also the bandwidth sharing among flows of the same protocol going through the same bottleneck, but different routing paths. Finally, we have the *"multiple bottlenecks"* scenario where the relative bandwidth for a flow traversing multiple bottlenecks is explored.

2.7.4 The state of the suite

A few words on the state of the evaluation suite are necessary. The TCP evaluation suite is the outcome of a "round-table" meeting in 2007 on TCP evaluation, and is described by Floyd et al. [17]. While the idea of creating a standardized test suite to evaluate TCP is a good one, there are a number of details that were not really thought through in the beginning. David Hayes picked up on the suite eventually and started work on improving the suite. The evaluation suite has been constantly updated while we have been working on this thesis. We have been working on a snapshot of the suite which we received from Hayes personally. At the moment, the project is at a halt because of a number of issues with the suite. Unfortunately we did not know this when we started working on this thesis. This section is based on conversations with Hayes.

The main problem with the suite is the way that congestion is measured. Each test defines a level of congestion measured in average drop rate. This means that traffic (from tmix) will have to be scaled in such a way that the test reaches a target loss rate. For instance, the target loss rate for the access link basic scenario with moderate congestion is 1.5%.

The problem with doing it this way is that if the network is always congested, packets will be retransmitted, which adds to the traffic, and congests the network further. After a while a network collapse will occur. This is a state in which hardly any traffic gets through the network because of the enormous amount of traffic. The evaluation suite defines an average drop rate for each scenario, which means that there will have to be quite

a lot of traffic, and that the network is always on the verge of collapsing. Figure 2.9 shows this network collapse with the three levels of congestion that each basic scenario defines marked on the curve. The network at this point is not very stable.

Tmix traffic also makes defining congestion by an average drop rate difficult. This is because the traffic constantly changes along the run of a simulation. At times there might be high peaks of traffic that is quite different to the rest of the traffic. For short simulations, it is possible to find a scale for tmix, so that the average drop rate of the run matches the target rate. However if the simulation is run for a long time, it is not possible to find such a scale, because at some point along the simulation there will most likely be a peak of traffic so high that the network will collapse and not recover from it. Avoiding the peak by setting the scale lower will result in the average drop rate being too low. According to Hayes, the suite will have to go through some drastic changes when it comes to defining congestion levels before the suite can be used for what it was originally intended: evaluation of TCP extensions.

The suite defines congestion by a target drop rate, but it also specifies that the output of each test should be the average drop rate of the central link. This in itself is quite weird, because if the drop rate metric does not match the target drop rate, then the test is not implemented as it was specified in the suite. This is also quite difficult when it comes to running the suite with different TCP variants, because when different TCP variants are used, different drop rates are measured, and all of them cannot meet the target drop rate. While the suite is a bit unclear about this, it is possible to compare the relative loss between different TCP variants as long as a "default" variant has been decided. The scale and target drop rate could be based on the "default" variant.

Through talks with Hayes we now know that the evaluation suite in its current state cannot be used as a tool to evaluate TCP; however there is no reason while the test cases cannot be used to compare network simulators as long as a proper scale is set, and the network does not collapse.

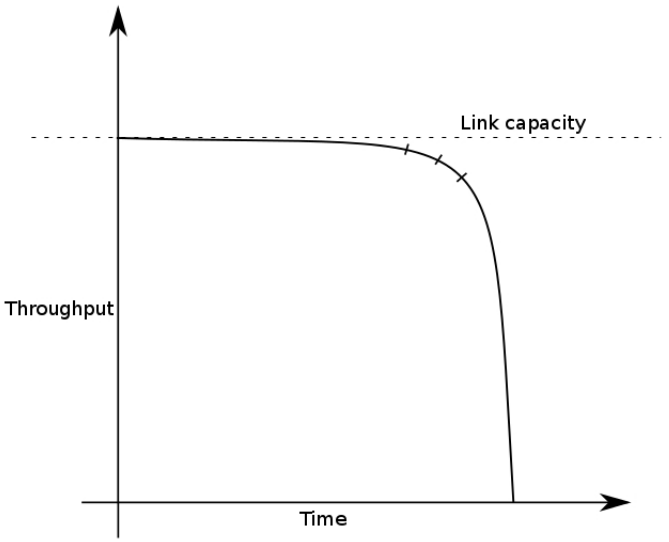


Figure 2.9: This graph shows network collapse due to high congestion. The throughput is close to the link capacity for a long time, but after a while the traffic is too much for the network to handle, and the throughput stoops. This is known as a network collapse. The three marks on the edge of the graph represent the three levels of congestion.

Chapter 3

The TCP evaluation suite implementation

In this chapter we will go through the TCP evaluation suite implementation for each of three simulators. The ns-2 version of the TCP evaluation suite [9] is available online and has been developed by David Hayes. The draft where the TCP evaluation suite is described will hereby be referred to as *the draft*. In this thesis we have used the TCP evaluation suite as a basic set of test cases that we can use to compare network simulators. Whenever the ns-2 version of the suite differs from the draft, we have tried to make our simulations as close to the ns-2 version as possible to make the results comparable.

As mentioned in the section about the draft, the TCP evaluation suite is not yet applicable as a tool to evaluate TCP due to a number of problems. The same goes for the ns-2 version of it. None the less, the ns-2 version of the tool should still be useful for us to evaluate and compare network simulators.

3.1 ns-2

This section will briefly introduce the ns-2 network simulator, as well as describe the ns-2 version of the TCP evaluation suite, with focus on the differences between the draft and the ns-2 implementation of it.

3.1.1 Introduction

ns-2 is an event-driven simulator, which has been proved useful in studying the dynamic nature of communication networks [23]. Due to its flexibility and modular nature, ns-2 quickly became a success in the network research community after its release in 1989. For many years the ns-2 network simulation tool has been the de-facto standard for academic research into networking protocols and communication methods according to [35], and a simple search on the web, both on Google[4] and on the ACM Digital Library[1], gives an enormous amount of results for ns-2 compared to other network simulators.

ns-2 consists of two key languages: C++ and Object-oriented Tool Command Language (OTcl). The internal mechanism of each module is written in C++, while OTcl is used to assemble and configure modules, as well as to schedule events.

ns-2 has a vast number of simulation models available online, and most research requires models which are beyond the scope of the built-in ns-2 modules. Incorporating these modules into ns-2 requires profound understanding of ns-2 architecture, and may feel like daunting task for ns-2 beginners. ns-2 has been criticized of being difficult to use, which is one of the reasons we see alternative network simulators today. Issariyakul et al. [23] says that *"...the formal documentation of ns-2 is mainly written as a reference book, and does not provide much information for beginners. The lack of guidelines for extending ns-2 is perhaps the greatest obstacle, which discourages numerous researchers from using ns-2"*.

3.1.2 The evaluation suite implementation

The ns-2 evaluation suite follows the draft described in Section 2.7 as far as possible. The main parts of the ns-2 version are implemented as described in the draft, but there are still several minor differences between the draft and the actual ns-2 implementation. The reason for why the ns-2 implementation differs from the draft is because it is currently a work in progress and therefore constantly changes. The snapshot that we have of the draft probably differs from the snapshot of the draft that the ns-2 implementation is based on. The ns-2 version of the suite can be downloaded from http://caia.swin.edu.au/ngen/tcptestsuite/tools.html. The connection vector files used can also be downloaded from the same location.

The suite is implemented in a very modular way, making it easy to add new tests to the implementation, as well as adding new TCP variants to it. The ns-2 implementation does not include every test described in the draft, but includes the following ones: the basic scenarios, the delay-throughput tradeoff scenario, impact on standard TCP, ramp-up time and multiple bottlenecks.

Every scenario in the ns-2 implementation uses the same set of connection vector files. To change the amount of traffic, they scale the traffic generated by tmix. This means that the time at which a connection is scheduled to start is *time * scale* instead of just *time*. This makes it possible to either spread out the connections, or to concentrate them. We have followed this example and added the possibility of scaling traffic in ns-3, and we have also built in this functionality in our OMNeT++ version of tmix. Each basic scenario is run three times with three different levels of congestion: uncongested, mild congestion, and moderate congestion. The draft does not specify a value for the uncongested run. In the ns-2 version, uncongested is twice the scale of mild congestion. For the delay-throughput tradeoff scenario, the level of traffic is specified to have moderate congestion when the buffer size is 100% of the bandwidth-delay product for a 100ms flow. Table 3.1 contains all the scales used in ns-2.

Scenario	uncongested	mild congestion	moderate congestion
Data center	0.604338	0.302169	0.285021
Access link	5.959628	2.979814	2.774501
Trans-oceanic link	0.36187	0.180935	0.169284
Geostationary satellite	25.500172	12.750086	11.993416
Wireless access	272.07901	136.039505	120.510034
Dial-up link	25328.998874	12664.499437	4755.809158
Delay/throughput	5.959628	2.979814	2.774501

Table 3.1: Tmix scales for all the basic scenarios and the delay-throughput scenario. The basic scenarios are run three times with different levels of congestion, while the delay-throughput is always run with the same amount of traffic

The total simulation time for each scenario is slightly different in the ns-2 implementation compared the times found in the draft. The warm-up time also differs from the draft, and there is an additional prefill time that is not described in the draft at all. At the beginning of each simulation, there is a prefill time, and after the prefill time, there is a warm-up time that is equally long. The prefill time is denoted *prefill_t*, and in addition there is a prefill start interval denoted *prefill_si*. *prefill_si* is calculated to be the maximum RTT of each scenario. Let's consider the access link scenario for instance. The maximum queue size is 102ms, and each packet will go through two queues. The longest path between two nodes is the path between node 3 and node 6, which has a total propagation delay of 100ms. The central link has a propagation delay of 2ms. This gives us a total one way maximum delay of 306ms, which gives us a maximum RTT of 612ms. In the prefill period, the start of connections is accelerated. This acceleration is decided by the start interval. What this means is that the connections that are supposed to start between [0, prefill_t], will instead start between [prefill_t - prefill_si, prefill_t]. For the access link scenario, where prefill_t is 30 seconds, and prefill_si is 0.612 seconds, this means that every connection that was supposed to start between [0, 30], will now be concentrated and started between [29.388, 30] instead. After the prefill period ends, there is a warm-up time equal to prefill_t to let the network stabilize before taking measurements. The actual simulation times for each scenario is shown in Table 3.2.

Through talks with David Hayes, we have learned that this way of doing the prefill is a temporary solution, and they quickly changed the way they calculated the prefill. What they did in the newer version was to calculate prefill_t and prefill_si in such a way that the network just barely reaches its target congestion level before the warm-up period starts.

The queue sizes also differ slightly from the description of the suite. These sizes are measured in ms, and are summarized in Table 3.3.

All in all, the salient features of the suite are implemented as described in the draft. However there are a quite a few differences in the configuration of each scenario, some of which will have a great impact on

Scenario	simulation time	warm-up and prefill	total time
Data center	55	18	91
Access link	300	30	360
Trans-oceanic link	60	20	100
Geostationary satellite	700	20	740
Wireless access	2000	20	2040
Dial-up link	20000	100	20200
Delay/throughput	300	30	360

Table 3.2: The simulation times for each scenario. The total time is equal to simulation time + warm-up + prefill

Scenario	buffer size L->R	buffer size R->L
Data center	22	22
Access link	102	102
Trans-oceanic link	232	232
Geostationary satellite	702	702
Wireless access	102	102
Dial-up link	1250	1250

Table 3.3: The buffer size in milliseconds for each basic scenario

the simulation. For instance, running a simulation for 20000 seconds is very different to running the simulation for 2000 seconds, especially when the tmix scale was calculated for the 20000 second run. In ns-3 and OMNeT++ we try to follow ns-2 as closely as possible.

3.2 ns-3

In this section we will first give a brief introduction to the ns-3 simulator, before we give a detailed description of the ns-3 version of the TCP evaluation suite. This section will also describe tmix for ns-3, as well as the modifications to it that we have had to do.

3.2.1 Introduction

In 2005 a number of researchers started developing a new network simulator to replace the ns-2 network simulator [35]. The tool was designed to replace ns-2, thus named ns-3. ns-3 however is not a newer version of ns-2, but rather a complete stand-alone network simulator based on ns-2.

One of the goals of ns-3 was to improve the realism of the models. There exist a number of high-level modelling languages and simulation-specific programming paradigms adapted and specialized for network simulators. This yields certain advantages, such as easily being able to develop models, but a high level of realism is typically not among the advantages. A high level of abstraction can cause results to diverge too much from reality, which is why ns-3 has chosen C++ as its programming language. C++

is a powerful programming language with many possibilities. One of the main reasons for going with C++, is because most protocol stack implementations are implemented in C, thus making them easier to incorporate into ns-3 simulations. Another reason to make a new stand-alone network simulator instead of making a new version of ns-2 is because of software maintenance. ns-3 (as well as ns-2) is not commercially supported, but open-source. This means that a lot of different people are contributing to the code-base. One of the main points of ns-3 has been to enforce a strict coding standard, as well as a code review process and a test infrastructure to ensure a high quality in all software added to the tool. This was not the case with ns-2 which made the tool grow considerably over time and led users to lose confidence in the results.

ns-3 models several objects from the real world. Below are a few examples of the models in ns-3:

- A *Node* represents end-points as well as routers, hubs and switches in the network. There is no difference between the nodes in a network. Computation power for instance is not taken into consideration. Hardware is also not a part of the model.

- A *Device* represents the physical device that connects a node to a communication channel. A switch for instance, may have several Ethernet network devices, while a wireless access point includes an 802.11 device on which packets may arrive wirelessly.

- *Communication channels* connect to network devices and represent a medium on which data may be transmitted. A communication channel could represent a fibre-optic link, a twisted pair cable, or the wireless spectrum for wireless communication.

- *Communication protocols* include common Internet protocols such as Ethernet, 802.11, TCP or UDP for instance. Many of the typical Internet protocols found in the TCP/IP stack are part of the ns-3 core. In ns-3 it is also possible to run real-world protocol stacks (such as the ones found in Linux or in FreeBSD) through the network simulation cradle (NSC).

- *Network packets* represent packets being sent on the network. Each packet usually contains one or more protocol headers, as well as a payload of data that depends on the application being used.

- *Applications* are modelled as well. What is the point of building a network if there is no traffic? The applications found in ns-3 are typically quite simple, for instance the *OnOffApplication* is an application that will send bursts of data at a given time interval.

Modelling elements from the real world is obviously necessary to create a simulator. In addition to the model itself, ns-3 includes several practical solutions which assist the user in both execution and analysis of a simulation: *attributes*, *helper objects*, and *trace sources*.

In most cases the user will have to change the behaviour of some of the models used in a simulation. Each model is defined by several *Attributes*. These attributes can easily be set by the user through public functions provided by the objects. The *PointToPointNetDevice* class for instance can be used with its default values, however if the user wants to change the behaviour of this device, that is easily done by functions provided by the class. ns-3 forces all models to be implemented with this attribute system.

Helper objects are used in ns-3 to assist the user in creating the simulation script. Consider a very common task in any network simulation: connecting two nodes to each other by a common communication channel. To do this, one typically needs to perform a number of core operations. First of all, we need at least two nodes, where each of the nodes has their own network device to which we can connect the channel. Each of the devices will have to be assigned a MAC address for the link-layer protocol to function properly. The nodes will need a protocol stack (such as the IP/TCP protocol stack) installed etc. This is a very common task, and ns-3 has a system to help perform such tasks more easily: helper objects. The point of these objects is to have the user quickly and easily perform common tasks without having to dig into the details. An example of a helper is the *PointToPointHelper* class. When using the *PointToPointHelper*, it is possible to set attributes for both the device and channel easily, such as the bandwidth and delay of the channel, and the maximum transmission unit of the device. Once these details have been decided, the *PointToPointHelper* will be installed on a group of nodes. Each of the nodes included in this group will get a *PointToPointNetDevice* installed, as well as a *PointToPointChannel* connecting every pair of nodes. The same goes for setting up a node with the TCP/IP protocol stack. There is a helper named *InternetStackHelper* which will install the entire stack on a group of nodes.

The goal of any simulation is to generate data that can later be analysed and presented. *Trace sources* facilitate the logging of data during an execution of a simulation. Trace sources are entities that signal certain events whenever they happen during the course of a simulation. The trace sources are not useful on their own, but must be connected to other pieces of code that actually do something useful with the information provided by the trace source. The entities that consume trace information are called *trace sinks*. Trace sources generate events, while trace sinks consume them. When a trace sink expresses interest in receiving trace events, it adds a callback to the list of callbacks held by the trace source. Whenever the trace source notices an event happening, it goes through this list of callbacks, and calls every trace sink registered. It is up to each and every trace sink to decide what they want to do when an event happens. There are several pre-defined packages in the core of ns-3 that use trace sources to generate data. For instance, there is a package that outputs PCAP[1] files based on a set of difference trace sources, and there is another package *flowmon* that

[1]PCAP [6] (packet capture) consists of an application programming interface for capturing network traffic. Monitoring software may observe the network and output every packet that passes through the network. These packets are printed in a PCAP file and can later be analysed using software such as Wireshark or tcpdump.

generates statistics on each flow in the simulation. Through the use of a combination of both pre-defined packages and custom made trace sinks, the user should easily be able to generate interesting data.

3.2.2 Tmix for ns-3

The core of ns-3 contains a number of basic traffic generators, but nothing like tmix, which is needed to make our ns-3 version of the evaluation suite comparable to the ns-2 version. ns-3 tmix[13] was developed by a group of students in a Google Summer of Code project in 2011[2].

The release of the tmix traffic generator for ns-3 was a very welcome addition to the simulator. However the generator is not perfect, the source code is filled with comments about missing functionality and weak implementation details that were to be fixed, but unfortunately never were before the release of it. As a result of this, we have had to do a bit of work to get the generator to do what we wanted. In addition to the traffic generator being a bit buggy, it is also completely lacking any documentation.

Tmix for ns-3 was originally developed for version 3.10 in 2011. Several new versions of ns-3 have been released since then. In version 3.13 the entire source code structure was redesigned. This made it necessary to reorganize the source code of tmix as well to adhere to the new ns-3 structure. The traffic generator consists of two modules: the tmix module, and the delaybox module. In the 3.10 version they were residing in folders that were removed in ns-3.13. To make it compile with ns-3.13, we made tmix and delaybox their own stand-alone modules. To make this work we had to change a lot of the header includes as many of the modules that the older version depended on were either not existing anymore, merged with other modules, or moved. In addition to fixing the dependencies, we also had to reorganize the code of tmix and delaybox into the new folder structure that ns-3.13 introduced. This means splitting the module, helper classes, documentation, examples and tests in their own folders. When we submitted the new ns-3.13 version of tmix, they released ns-3.14, which made tmix crash once again. Luckily, this was just a matter of renaming a few constants.

In ns-2, they have extended the functionality of tmix to make it more suitable for the TCP evaluation suite. They have, for instance, created an additional parameter *'m'* that defines the maximum segment size (MSS) of each connection vector. To do this in ns-3, we first had to extend the parser so that it read the *'m'* parameter, and then stored it in the connection vector structure together with the other parameters. When a tmix application is started, it creates a socket to communicate through. We modified the configuration of this socket so that the MSS was set correctly.

In ns-2, they have also extended the functionality of tmix so that it is possible to scale the traffic. We have done the same for ns-3 tmix. This was just a matter of setting the time at which a connection is due to start to *"time * scale"* instead of just *"time"*.

[2]Google Summer of Code is a global program that offers students the possibility of working on an open-source project together with a mentor.

Tmix includes a set of helper functions included in the file *tmix-topology.cc*. This makes it very easy to create a dumbbell topology where the nodes use the tmix traffic generator without having to dig into the implementation details. The problem with this helper was that each acceptor expected to only communicate with one initiator. In our evaluation suite however, every node on the left side communicates with every node on the right side.

The way that the communication was set up was not very intuitive. Tmix traffic is always set up between an initiator and acceptor. In the ns-3 version of tmix, when the initiator started a new connection, it would generate a new port for that connection. It would not use the new port to bind a new socket itself however. The initiator would call a function at the acceptor, declaring that the initiator would soon connect to the given port. The acceptor would then listen to a connection on that port, and accept only the first connection. The problem occurred when we tried to let one node act as an acceptor in several initiator/acceptor pairs. This was because the initiators allocated port numbers on their own. These port numbers conflicted with the port numbers that the other initiators allocated.

The best solution would be to completely redesign this, and handle the client/server communication a little differently. Instead of letting the initiators allocate ports for the server, they could rather allocate ports for themselves. The server could listen for connections on the same port throughout the entire run of the simulation, and create a thread for each incoming connection to handle it. This should be much simpler and more intuitive as well. We have chosen an alternate solution, which is easier to implement, but is really just a quick fix for the problem.

The tmix-topology helper sets up two routers and a pair of nodes communicating in the initialization phase. It has a function named *"NewPair"* that adds an extra pair of nodes: one on the left side, and one on the right side. This function sets up the nodes, as well as a tmix application that generates the traffic. We have created an additional function named *"AddAdditionalApplications"*. This does exactly the same as the *NewPair* function except for actually creating two new nodes. Instead it takes a pair of already existing nodes as input and creates a tmix application in both of them that communicate with each other. To make sure that the port numbers do not conflict with each other, a range of port numbers is set in the application. The port allocator will start at the lowest port available for the application, and never go above the port number limit.

All in all, the tmix version that we have received has been of great help in this thesis, although at times frustrating to use because of the lack of documentation.

3.2.3 The evaluation suite implementation

The ns-3 TCP evaluation suite has been implemented according to the description in section 2.7. Where the ns-2 version of the suite differs from the description of the suite, we have chosen to make our version as close to the ns-2 version as possible to make the results comparable. All the files of

the suite are bundled together in a folder named *eval* which is located in the *examples* folder of ns-3). This means that the TCP evaluation suite in ns-3 is implemented as an ns-3 example. This does not really have an impact on the suite, only on how the suite is compiled. For instructions on how to use the evaluation suite, see Appendix A.

Architecture

The test suite consists of a few components. For every topology there is a file building that topology. For instance there is a file for the dumbbell topology, named dumbbell.cc (and its header file dumbbell.h). In addition to the topology, there is another file (named after the scenario it describes) that configures a scenario. For instance there is a file named basic-scenario.cc that configures each of the basic scenarios. This means choosing the correct topology, choosing the bandwidth and delay of each link, and setting up the traffic and so on. The *utils* folder contains a few utilities, such as the timestamp-tag, which makes it possible to attach a timestamp to a packet, and a parser that can be used to parse simulation output. To easily use the evaluation suite, a main script has been included, *tcp-eval.sh*. This script accepts simple commands such as "./tcp-eval.sh accessLink" to start the access link scenario, as well as run the analysis script when the simulation is finished. This hides the implementation from the user.

Topology and protocol stack

The topology used in the simulation depends entirely on the scenario to be run. Most of the scenarios in the evaluation suite use a typical dumbbell topology. The basic setup for most of our tests, is described in a file named *dumbbell.cc*, which sets up a dumbbell topology using the *TmixTopology* class as help. The *TmixTopology* class is a sort of helper to quickly help the user setting up a standard topology for testing purposes. The topology always contains a central link, and it is possible to create pairs of nodes in addition to the central link, where one node is on the left side, and one node is on the right side. These nodes are all properly set up using the InternetStackHelper so that every node has a complete stack of protocols. In addition to the normal nodes and routers, the topology contains a DelayBox, which is required for tmix.

As our link-layer protocol, we use the very simple Point-to-Point Protocol (PPP). This is a very basic protocol that hardly contains any functionality at all. The reason for using such a simple protocol is to ensure that we do not fool TCP into thinking the network is congested. TCP will always treat lost packets as lost because of congestion, and as such will adjust its sending rate to avoid further congestion. We want to avoid this happening because of random errors in the link and physical layer, therefore we have this very simple link-layer protocol together with PPP channels that never drop packets, nor generate bit-errors.

The queues used for each device are all of the type *DropTailQueue* for most scenarios. The queue sizes are specified in the TCP evaluation suite,

but we use the values presented in the chapter about the ns-2 version of the suite. The topology builder class will take queue size in milliseconds as input, but will calculate the actual size in number of packets. This is done using the following formula:

$$\text{queue size} = \frac{C * T}{M} * \frac{1}{8}$$

where C is the central link's capacity, T is the maximum wait time for a packet, and M is the maximum packet size.

The network-layer protocol used is IPv4. The only other option really would be IPv6, but choosing either one should not make an impact on the simulation. The packets will be able to find their way through the network either way, and the packet overhead is not that different between the versions anyway. The protocol header in IPv6 is slightly larger than in IPv4, but this should not make a significant impact on the simulation, and even if it was significant, we use IPv4 for all three simulators. The network is configured in such a way that every endpoint on the left side has addresses of the form 10.1.0.0/16. The endpoints on the right side have addresses of the form 10.2.0.0/16, while the routers have addresses from the network mask 10.3.0.0/16. This means that our network is split into three subnets. The reason for this is that it makes it very easy to decide in which direction a packet is heading. If it is heading towards an address that starts with 10.1 for instance, it is heading towards the left side of the network. If it is heading towards an address that starts with 10.2, then it is heading towards the right side.

ns-3 has several TCP variants built-in: TCP with no congestion control, Tahoe, Reno, and NewReno. We have chosen to use Reno and NewReno in our simulations. TCP with no congestion control is very out-dated, as is TCP Tahoe. In addition to the built-in variants, ns-3 has the possibility of running real-world protocol stacks through the *Network Simulation Cradle* (NSC) [8]. NSC is a framework which allows real world TCP/IP stacks to be used inside a network simulator. ns-3 currently only supports the Linux stacks. The downside of running real-world protocol stacks, as opposed to running built-in protocol stacks, is that the real-world ones are much more complex, thus slowing down the simulations by quite a bit. We tried to run CUBIC through NSC in ns-3. Without changing anything but the protocol stack however, the simulation crashes after running for a very long time. Only the shortest running test (the dial-up link scenario) was able to complete using NSC, and this took quite a long time compared to running the built-in version.

Traffic generation

When we began working on this thesis, tmix was not yet released. We were looking for alternative ways to generate traffic that would look somewhat like the traffic that tmix generates in ns-2. In ns-3 there are a few basic applications that generate traffic, including simple bulk transfer applications, on-off applications (which send bursts of data at a given time

interval) etc. These are too simple and too predictable to be comparable to tmix.

We found a tool based on the Poisson Pareto Burst Process (PPBP) [16] for generating Internet traffic. The PPBP traffic generator aims to create a generator that matches certain attributes from real-life IP networks. We used this traffic generator for a while, and even though it is more realistic than the simple applications in the ns-3 core as it is less predictable, it is still difficult to configure the generator in such a way that the traffic reassembles traffic as generated by tmix.

During the summer 2012, tmix for ns-3 was released and we decided to use it. Traffic is described through a series of connection vector files, defined as a part of the topology definition. Each node on the left side communicates with every node on the right side and vice versa. This means that there are nine pairs of communicating nodes, and for each one there is a connection vector file specifying the traffic. All traffic flows through the central link. This is the same as for ns-2.

We use the same connection vector files that are used in ns-2. The problem however is that they are in the original connection vector format (as described in the ns-2 manual [27]), while tmix for ns-3 only supports the newer alternate connection vector format. Some time was spent trying to make the parser in ns-3 tmix compatible with both formats, but through private correspondence with Michele Weigle (who has been part of the tmix project for both ns-2 and ns-3) we have received a script that converts from the original format to the alternate format. Since the connection vector files in the ns-2 version of the suite contains the additional *'m'* parameter, we have extended the script to include this parameter as well.

Collecting and presenting statistics

The point of any simulation is to collect data and to present them in a meaningful way. For a description of the metrics to be collected see section 2.7. As mentioned in section 3.2.1, ns-3 has several trace sources available that will notify trace sinks whenever certain events occur. There are several pre-existing packages that produce statistics in ns-3. We have implemented this part of the simulation twice. We will describe both our solutions as they both give the correct results, however, the first solution is more complex, slows down the simulation, as well as requiring quite a lot of disk space, which is why we decided to redo it.

One of the reasons for the poor design in our first attempt was because we did not fully understand the tracing system in ns-3. This made us rely mostly on the pre-defined tracing packages. To generate our results, we made every PPP net device in the simulation generate pcap files. The pcap files were then parsed with the parser found in the *utils* folder. To parse the pcap files we used the libpcap package for developers. This contains a lot of functions and structures to ease the process of parsing pcap files. The parser is best described in pseudo code:

```
for (every packet in the pcap file)
```

```
read PPP header
check that the next header is an IP header

read ip header
figure out which way the packet is heading

add packet size to the total amount of bytes
```

The parser contains a main loop, which reads one by one packet from the file until there are no more packets. There are structs in the libpcap package that corresponds to the PPP header and the IP header, thus making it easy to access the fields of these headers. In the PPP header we check that the next header is in fact the IP header. Next we have to figure out in which direction the packet is heading. As the left and right side of the central link are each in their own subnet, this is easy to determine by matching the IP address in the IP header with the subnet mask of the two subnets. The size of the packet (found in the IP header), along with two more bytes for the PPP header, is added to a variable that keeps track of the amount of bytes we have counted so far in each direction.

One problem with analysing the simulation this way is that the pcap files themselves are useful, but only to a certain extent. We now know how many bytes that have passed in each direction, but the pcap file does not contain any information on the capacity of each link, the simulation time or the amount of packets that were dropped. Because of this, the parser also relies on several values that have to be passed to the parser somehow. What we did was to create a file named *stats.txt* where we print each of the values we need from the simulation in comma-separated values (CSV) format. This file was also parsed by the parser. Once all the intermediate results were parsed and calculated from the pcap file and the stats file, the final results could be calculated.

In addition to the pcap file, ns-3 also has the possibility of outputting some statistics on flows through the use of flow monitors. These flowmon files are written in xml and contain statistics such as the time at which the first packet in the flow was sent, the total amount of delay for the flow, transmitted packets, total amount of jitter and so on. Our first idea was that there are already good solutions out there for parsing xml files, such as xQuery, which is a query language for xml files. However, we decided against it as the amount of work to learn xQuery would probably outweigh the amount of work to just write a simple parser in C. Performance in this case is not a problem as the flowmon files are not that large anyway. The main loop of the parser goes through each flow in turn till there are no flows in the file. For every flow the following statistics are read by the parser: lostPackets, rxPackets, delaySum, txBytes, txPackets, timeLastTxPacket, timeFirstTxPacket. These are used to calculate statistics such as sending rate, good-put, the number of lost packets, delay and the number of successfully delivered packets.

At the end, after parsing the stats file, pcap file, and the flowmon file, the parser will generate the final statistics and output it in CSV format.

The parser was our first attempt at generating statistics for ns-3. However, as we learned more about the simulator, and especially learned more about the tracing system, this seemed like a much simpler and more elegant way of handling statistics. The trace sources that are generated by the queues are especially useful. Each queue will report each enqueue, dequeue, and drop event. One way to connect trace sinks to trace sources is through the use of the config system. The config system requires a full path to the needed trace source, as well as a trace sink to call when the trace source reports an event. The paths of the trace sources found in ns-3 are well documented in the ns-3 application programming interface (API) (although quite hard to find if you do not know where to look). As an example, see the code below:

```
std::stringstream ss;
ss <<
    "/NodeList/"
    << top->GetRouter(0)->GetId()
    << "/DeviceList/*/$ns3::DelayBoxNetDevice/TxQueue/Drop";
  Config::Connect (ss.str(), MakeCallback (&DroppedPacketSink));
```

At the beginning of the path is a global list of all nodes, the "NodeList", followed by a node ID. The config system accepts wildcards, so that it is possible to choose all nodes by substituting the node ID by an asterisk. Each node contains a DeviceList containing all the devices of the node. The next part of the path is an asterisk, which means that we want to look through every network device of the node. The next segment of the path starts with the '$' character. This character indicates that a *GetObject()* call should be made looking for the type that follows. In our case, this means that we should look through every network device in our router looking for network devices of the type DelayBoxNetDevice. For every device found, we access the *TxQueue* attribute (the transmit queue) and finally the trace source named *Drop*.

There is a number of interesting trace sources that we can use to calculate statistics for the central link. The queues of the central routers has three trace sources: *Drop*, *Enqueue*, and *Dequeue*. These are all very useful. In addition to this we use the *MacTx* trace source of all the edge nodes. This is used to find the total number of transmitted packets in each direction.

Whenever a packet is dequeued onto the central link, we know that it is going to reach its final destination. We can be certain of this because the central link will not drop packets, and the link connecting the destination node will never be congested as the capacity of the link is always greater than, or equal to the capacity of the central link. Because of this, we can use the dequeue event on the routers to count the amount of successfully delivered packets, as well as the amount of successfully delivered bytes. The sum of all delivered bytes, divided on the simulation time gives us the average throughput of our simulation. We can count the number of times packets were dropped from these devices as well to calculate the drop rate of the central link.

Queuing delay is a little trickier. The idea is that we attach a timestamp

to each packet as they enter a queue, and when they dequeue from the queue we can read the timestamp to see how long the packet waited in the queue. The packet should go through two queues along the path to its destination. When it arrives at the first router, it will wait in queue before it is being transmitted on the central link. When it arrives at the second router, it will wait in queue to be transmitted onto the edge link. Note that this second number will always be zero, or very close to zero. If we calculate the sum of all queuing delays, and then divide this by half of the total amount of dequeuing events (half because every packet generates two dequeuing events), then this should give the average queuing delay of each packet.

ns-3 has the possibility of adding tags to packets. All tags must derive from a common class *Tag*. There is an example included with ns-3 that extends this tag so that the tag includes a timestamp. We use this class, named *TimestampTag*, to attach a tag with a timestamp to our packets. This class is found in the *utils* folder.

We have just described two ways to collect the statistics we need from our simulations. The first way was very complex and all in all quite unnecessary, while the second is much simpler and more elegant. The flowmon parser is not needed either, as we only analyse the statistics from the central link.

Improvements

We would have liked to test other TCP variants as well, such as CUBIC. The problem however is that we have not been able to run NSC properly. First of all, NSC increases the run-time of each simulation by an enormous amount, which makes it difficult to work with. Secondly, NSC crashes after having run for a very long time. We believe that the simulation crashes because NSC uses too many resources, and possibly because of memory leaks. We have not been able to verify this however; these are just our initial thoughts.

Our implementation of the TCP evaluation suite is not particularly easy to use in its current state. We were planning in the beginning to make it very user-friendly, but as we learned at a later stage that the TCP evaluation suite as it is today has a number of flaws, we did not spend too much time on this as we knew we were not going to submit this code anyway.

3.3 OMNeT++

In this section we will first give a brief introduction to the OMNeT++ simulator, before we give a detailed description of the OMNeT++ version of the TCP evaluation suite. This section will also describe our implementation of the tmix traffic generator for OMNeT++.

3.3.1 Introduction

This section is a summary of the most vital information from the OMNeT++ user manual [10]. OMNeT++ is often quoted as a network simulator[35], but is in fact a generic architecture that can be used in a number of problem domains, for instance: network modelling, protocol modelling, modelling of multiprocessors, and validation of hardware architectures amongst others. OMNeT++ provides infrastructure and tools for writing different kinds of simulations. It also provides a number of utilities to use together with the simulation, e.g., classes for random number generation, statistics collection and topology discovery.

OMNeT++ in itself is quite basic and will in most cases be used together with other frameworks. OMNeT++ provides a very basic architecture for designing and connecting components. A simulation model is assembled from several reusable components, called *modules*. These modules are the basic building blocks of any model and can be connected to each other much in the same way that one might connect LEGO-blocks. Modules are connected via *gates* (the input and output interfaces of a module) and communicate with each other by passing messages. These messages may pass through a pre-defined route (as defined by the gate connections), or they may be sent directly to a destination gate without passing through connections. The latter may be useful in simulating wireless connections for instance. A module may also send messages to itself. This is used to schedule events within a module.

The smallest building block in an OMNeT++ simulation is called a *simple module*. Simple modules may be combined to create *compound modules*. An example of this could be implementing each component of the TCP/IP protocol stack as their own module, and combining all these modules into a larger compound module, representing a node on the network containing the entire TCP/IP stack. In Figure 3.1, we can see a compound module named StandardNode from the INET framework. It consists of a number of modules, some of which are connected, and some of which are not. The entire TCP/IP stack is represented in the figure. At the bottom we have a number of different link-layer protocols, wireless LAN, Ethernet, PPP and optional extensions. These have connections to the gates of StandardNode, which are represented by the lines going from the link-layer protocols to the bottom. Note that the only way to communicate with this node is to go through the gates at the bottom and through the link-layer protocols. All the link-layer protocols are connected to the network layer, which includes the IP protocol. The network layer is connected to three transport-layer protocols: tcp, udp and sctp. These are again connected to application at the top. In addition to the protocols, the StandardNode also contains a few other modules, which we will not go into details about here.

Simple modules are written in C++. This is a level of abstraction that makes it possible to write very detailed modules (i.e., close to real world implementations), while still having all the power of a full object oriented programming language. A simulation model in OMNeT++ is referred to

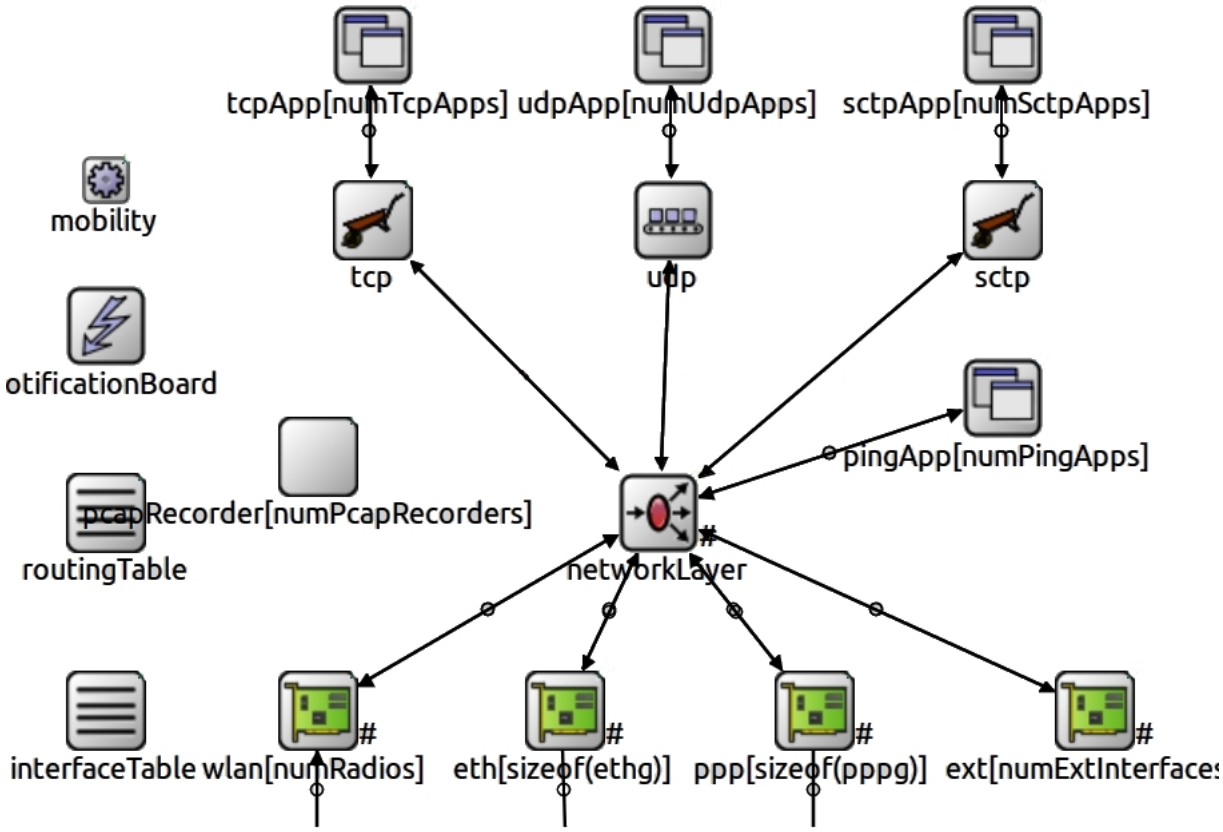


Figure 3.1: TCP/IP stack in OMNeT++

as a *network*. A network consists of hierarchical nested modules, where the top level module is called the *system module*.

While the module functionality is written in C++, the module structure, and its sections, are described in OMNeT++'s NED language. This is a very simple scripting language that describes for instance how many gates a module should have, how each gate is connected to other gates and which submodules a compound module consist of. Each module in OMNeT++ may contain several sections that together describe the module, all of them are optional:

- types
- parameters
- gates
- submodules
- connections

The types defined in a NED file are really just local variables, used within the NED file. Parameters are variables that belong to a module. They can be used to specify attributes such as the protocol to be used, the number of nodes in a simulation, packet length and so on. The NED file also specifies gates as already mentioned. These are the connection points between modules in a simulation model. When defining a compound module, the submodules will have to be defined as well. Lastly, connections between gates are also defined using the NED language. When specifying a connection, we may define simple attributes along with the connection, such as delay and bandwidth, but there are also more complex connections including loss rate etc.

Configuration and input data for the simulation are in a configuration file, normally named omnetpp.ini. This file is line oriented and consists of section headers, or key-value lines. When running an OMNeT++ simulation, you get to choose from the different sections defined in the configuration file. The most commonly used section is the General section, which is the one used by default. This section will usually contain a base configuration for all your simulations, while the other sections will contain the specifics of each single simulation. In each section there are key-value lines, where the key is the path of a module parameter, and value is the value of the parameter. For instance *"Network.server.transactionsPerSecond = 100"* will find the Network named *Network*, then a node of some sort named *server*, and then the *transactionsPerSecond* parameter of the server. The configuration file accepts wildcard patterns as well, making it easy to configure groups of similar modules. A single asterisk matches zero or more characters except for dots (which means that it will only match a single module), while two asterisks match zero or more characters including dots (which means that it may match several modules). For instance, *'**.tcpType = "TCPReno"'* will match any network and any node

and look for a parameter named tcpType. It will then set this type to *"TCPReno"* in every module that contains this parameter.

Collecting statistics is an important part of any simulation. OMNeT++ provides built-in support for recording simulation results, via *output vectors* or *output scalars*. Output vectors are time series data, recorded from simple modules or channels. Examples of output vectors may include the round-trip-time (RTT) of each packet between a pair of nodes, the queuing time of each packet in a queue or packet drops. Output scalars on the other hand, are summary results. These scalars are computed while the simulation runs, and written out when the simulation ends. These scalars are typically statistical summaries of several fields. For instance, output scalars may be results such as the number of packets dropped at a queue, the average queuing time for a flow, the longest RTT ever measured, the average link utilization and so on.

There are two fundamental ways of recording statistics in OMNeT++. The first is based on the signal mechanism, which was introduced in OMNeT++ 4.1, while the second is based on directly recording statistics in the C++ code by using the simulation library. The signal mechanism is the preferred method of handling statistics according to the manual. Signals are emitted by components. They are identified by signal names. Other modules may listen to signals, and act whenever a signal of the appropriate type occurs, or the signals may be used to generate statistics. When a signal is emitted, it can carry a value with it. The value may be a few selected primitive types, such as integers, or it may carry an object pointer, such as a pointer to a cPacket.

Statistics are declared in a module's NED file by the *@statistics* property. A statistic is based on one or more signals. The statistic may be very simple by just outputting a simple signal, thus outputting a vector containing all of the signals from the same signal ID. This could for instance be a vector containing all the packets that were dropped. The statistic may be more complex as well by generating aggregate statistics , such as the sum of all lost bytes in the simulation.

In the IDE included with OMNeT++, there is an analysis tool that is very helpful. This tool reads OMNeT++ output files and has the possibility of combining chosen values into datasets, on which it is possible to perform certain actions. These actions include computing the average of all sums, multiplying values, dividing values and so on. In addition to mathematical functions, it is possible to output datasets in (amongst others) comma-separated values (CSV) format, which is useful for further analysis. According to the manual, the IDE's analysis tool lacks a bit in handling scalars, and suggests using other tools such as R [12], which is a tool for statistical computing. The signal mechanism, along with the analysis tool found in the IDE makes it relatively easy to work with statistics in OMNeT++.

The basic simulation framework however is not enough for our purpose. In addition to the OMNeT++ simulator, we have used the INET framework [5] to add Internet functionality to our simulations. The INET framework contains the most common Internet protocols, such as IPv4,

IPv6, UDP, TCP and several simple application models which enhance the basic OMNeT++ framework. It contains a number of link-layer protocols such as PPP, Ethernet and 802.11, as well as routing configurators to easily route packets along the network. These two frameworks together, as well as a tmix module, which we have implemented ourselves, are enough to create the TCP evaluation suite for OMNeT++.

3.3.2 Tmix for OMNeT++

The INET framework contains a number of quite basic traffic generators, but nothing like tmix, which is what we need to make the OMNeT++ simulation comparable to the other simulators. This section describes our implementation of tmix for OMNeT++. It is based on tmix for ns-2 and ns-3, as well as the basic applications included in the INET framework.

We have extended the functionality of tmix to include the *'m'* parameter, which gives the maximum segment size (MSS) of each connection. We have also extended tmix so that we are able to scale our traffic, like they have done in the NS2 evaluation suite.

Architecture

First of all, a brief overview of the tmix architecture for OMNeT++ will be given. Every host that is part of a simulation using tmix must include a very simple tmix server. This server accepts incoming tmix messages, and acts upon the information that this tmix message contains. All hosts must also include a tmix client, which is responsible for generating tmix messages. Note that both the initiator and acceptor must include both the server and the client part of tmix. The path between an initiator-acceptor pair must go through a special router, a so-called DelayRouter. This DelayRouter extends the functionality of a standard router by including a module named delaybox. The delaybox will delay traffic in such a way that the minimum round-trip time (RTT) specified in each connection vector is ensured, which means that it delays each packet by half of the minimum RTT. A packet travelling between node A and node B, and back against, should therefore be delayed by half of the minimum RTT twice, which means that a packet will never arrive back at the sender faster than the minimum RTT specifies. Figure 3.2 shows the basic components of tmix and how they are connected. It doesn't matter where the DelayRouter is placed, as long as every single packet that is sent between the initiator and acceptor goes through it. Now that we have our basic architecture in place, we will describe each component in detail.

The tmix message

The tmix application sends tmix messages back and forth between the initiator and the acceptor. We will quickly mention how these messages look like, because it is vital in understanding how the server works. The OMNeT++ definition of our packet looks like this:

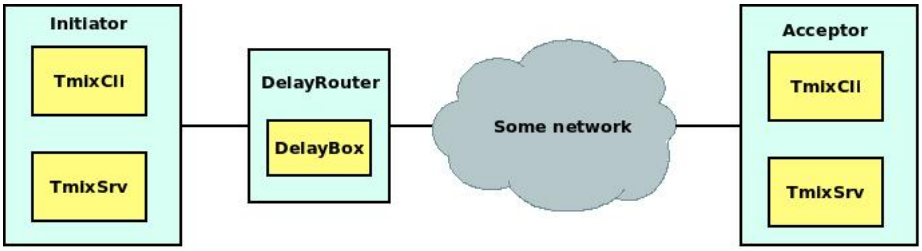


Figure 3.2: Basic overview of tmix components

```
packet TmixMsg
{
    int expectedReplyLength;
    double replyDelay;
    bool serverClose;
    bool serverSendFin;
    double serverWaitBeforeFin;
    int totalBytes;
    simtime_t totalWaitTime;
    int numberOfAdus;
}
```

This message definition is an extension of the cPacket from the OMNeT++ framework. To set the actual size of a packet we can use the setByteLength() method, inherited from cPacket. Note that other attributes in the message does not count towards a larger packet, they are just metadata describing the packet, but not actually part of it.

The tmix server

The server is very simple. Each new incoming connection gets its own thread on the server to handle the connection. These threads are not real threads, but are simulated threads that make the server act like a real threaded server. The basic setup for this server is based on the TCPSrvHostApp and TCPGenericSrvThread from the INET framework. There is hardly any logic in the server at all. Upon the reception of a packet, the server will go through the attributes in the packet and act accordingly. The first two attributes tell the server that it should reply with an application data unit (ADU). *ExpectedReplyLength* is the size of the ADU, while *replyDelay* tells the server to wait a while before replying. *ServerClose* is used to tell the server that the client does not expect any more ADUs, and will close the connection after receiving an acknowledgement (ACK) to the current ADU. If *serverSendFin* is set, it means that the server should be the one closing the connection. This attribute is used together with *serverWaitBeforeFin*, which tells the server how long it should wait before closing the connection. As we can see from this short summary, is that the server hardly does anything. Most of the logic is placed in the client, which must go through the connection vector to plan how both the client should

behave, and also how the server should behave. Below is a description in pseudocode, illustrating how the server works.

```
--> receives packet

if (serverClose)
  Do nothing

if (expectedReplyLength != 0)
  CREATE tmix msg
    SET bytelength to be 'expectedReplyLength'
  SCHEDULE send event after 'replyDelay μs'

if (serverFin)
  SCHEDULE close event after 'serverWaitBeforeFin μs'
```

The tmix client

The client is slightly more complex than the server. The client, as well as the server, creates a new thread for every connection (again, not a real thread, but a simulated thread that acts like real threads). On initialization in the main thread, the client parses a connection vector file specified in the NED file for the module. Tmix for OMNeT++ expects to find an environment variable named VECTORFILES, which should contain the path to the folder where each vector file is found. This environment variable makes it easier to run the simulation on different computers, and also makes the NED file neater, by only having to specify the file name, without its path.

This parsing will generate a ConnectionVector object for each connection vector in the file. Note that it will not schedule connections that would normally start after the simulation ends. This significantly reduces the time it takes to start the simulation, as well as the memory usage. The ConnectionVector object contains all the information one would need to know about a single connection vector: some general information about the connection (such as the minimum RTT and the loss rate of the connection), a list of ADUs and their attributes, as well as some additional control information needed by the client to function properly. This includes information about which ADUs have been sent, and how far along the list of ADUs we have come and so on. When the parser finds a new connection vector, it will create a new thread, and pass the ConnectionVector object as argument to the new thread. Before it gives control to the child thread, it will first create a socket for the child to communicate through, and pass a pointer to the socket as argument as well. The main thread keeps track of all sockets, and decides what to do whenever an incoming message arrives (i.e., figures out which thread the socket belongs to, and forwards the incoming message to the correct thread). The main thread will schedule the child to start at the time when the connection vector starts.

In the initialization phase of the child, it will first of all bind the socket that it got from the parent. Then it will register this connection at the

DelayRouter (more on this later), and lastly it will connect to the connect address given as parameter in the NED file.

Upon connection establishment, the socket API in the INET framework, will call the established() method. After connection establishment, the client will start scheduling ADUs. Whether the client is an initiator or an acceptor is given as a parameter in the NED file (remember that both the initiator and acceptor includes both the client and the server part of tmix). If the client is an acceptor, it will only schedule ADUs from concurrent connections, and ignore all the sequential connections. If the client is an initiator, it will schedule ADUs from all connection vectors. On established(), the client will find the first ADU that the initiator/acceptor should send, and schedule it (for the initiator this will always be the first ADU, for the acceptor we might have to look through some ADUs to find the first one in a concurrent connection). The scheduling of ADUs is best explained in pseudocode:

```
--> Established()
  if (concurrent connection)
  {
    if (acceptor)
      Find first acceptor ADU
      SCHEDULE send event for the ADU

    else if (initiator)
      Find first initiator ADU
      SCHEDULE send event for the ADU
  }
  else if (sequential connection)
    SCHEDULE first ADU
```

When we schedule an ADU, what we really do is make the module create a message that it sends to itself at the correct time. When the self message arrives at the module, it will schedule the next ADU to be sent. We have the possibility of giving the message an identifier. By doing this, we can tell the module whether the ADU to be sent is part of either a concurrent, or a sequential connection. When a self message arrives, the client will plan ahead, and schedule more ADUs. Again, this is best explained in pseudocode:

```
--> onSelfMsg()
  if (MsgKind == Concurrent) {
    Send this ADU
    Schedule next ADU
  }
  else if (MsgKind == Sequential) {
    if (we have already sent the last ADU)
      Do nothing

    else if (this is the last ADU)
      SET serverClose
```

```
    SEND this ADU

  else if (we should send two packets in a row)
    SEND this ADU
    SCHEDULE next ADU

  else if (the next ADU is the last ADU)
    SET serverSendFin
    SET serverWaitBeforeFin to zero
    SEND this ADU

  else if (the two next ADUs are acceptor ADUs)
    SET serverSendFin
    SET serverWaitBeforeFin
    SEND this ADU
  else
    SEND this ADU
}
```

In the ConnectionVector object we keep a bit of additional information that is not really part of the connection vector, but is rather used as a part of the client algorithm. This includes an index to tell how far along the ADUs we have come. Whenever we send a message, we increase this index as well, so that we know which ADU is the next one to send. When we receive a self message, handling a concurrent connection is very easy, as the client is not depending on the server it is communicating with. As we do not have to wait for ADUs from the server, we can just send the current ADU, and schedule the next ADU right away.

Sequential connections are a little more complex (note that only initiators schedule ADUs from sequential connections). The first thing we check is whether our index has been increased beyond the number of ADUs we should send. If so, we do not have to do anything, as the last ADU has already been sent. If the current ADU happens to have the last index, we can send the ADU, and tell the server that the current ADU is the last ADU and that we do not expect to get a reply (other than the ACK, but that does not involve the server *application*, only the TCP module). If there are two packets in a row that are both sent from the initiator, we send the first one, and schedule the next one. When we are getting near the end of a connection vector, we will have to plan ahead to see how we should close the connection. If the next ADU is the last ADU, then we know that the server should close the connection right away, so that is what we tell it to do. The last conditional statement checks whether there are two ADUs in a row, both from the acceptor. This only happens in one special case. The first of these two ADUs is a normal one, while the second ADU is a special ADU where the size if zero. This second ADU means that the server should close the connection, and this ADU also specifies a wait time. This means that the server should send the first normal ADU, and then it should wait the time specified in the second ADU before it closes the connection. So

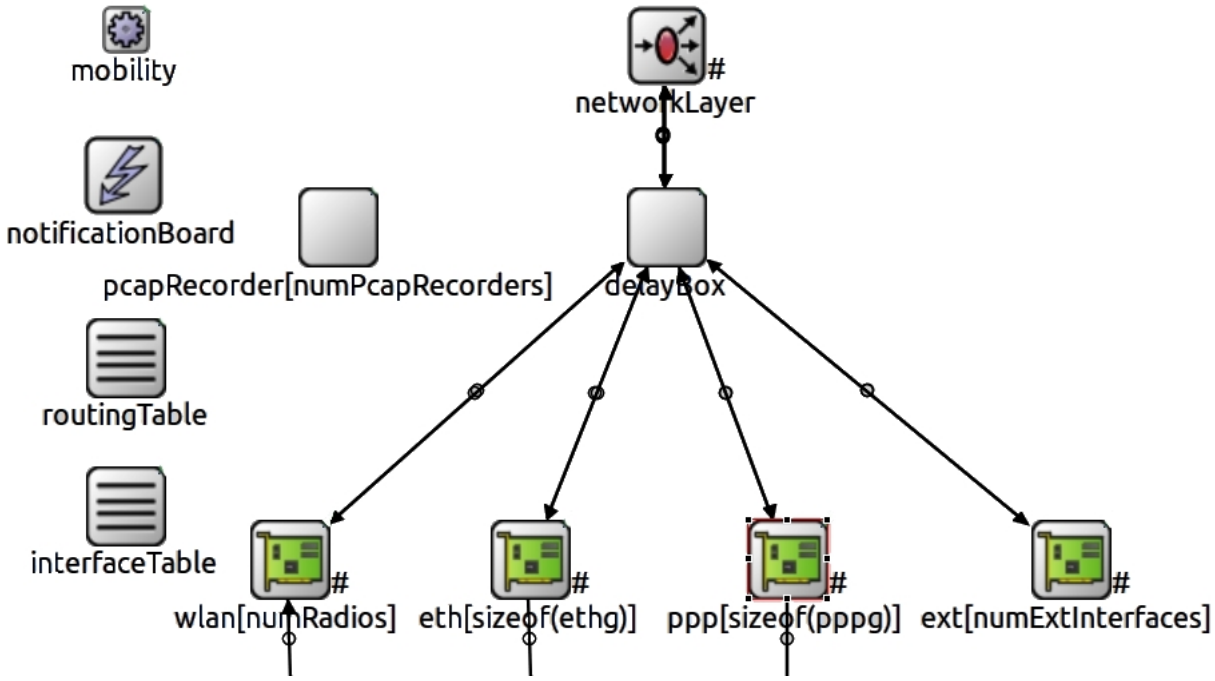


Figure 3.3: A DelayRouter and its submodules

whenever we find two ADUs in a row, both from the acceptor, we tell the server to send one normal ADU, and then wait a while before closing the connection.

The tmix DelayRouter

The last component in our tmix implementation is the DelayRouter. The DelayRouter is a bit different from the other components. While the server and client work exclusively in the application layer, the router will have to work in other layers as well. By default, a message arriving at a router, will first go through either one of its link-layer protocols, before being forwarded to the network layer. The network layer will decide where to forward the packet, before the packet goes down through the protocol stack again. The DelayRouter on the other hand, will also have to figure out which connection this message belongs to, and then delay the message an appropriate amount of time, before sending it further. The module that implements this functionality is called the delaybox. The DelayRouter is shown in Figure 3.3. In this figure you can see that an additional delaybox module has been placed between the link-layer protocols, and the network layer.

As we know, TCP identifies a connection with two pairs of sockets: the local socket, and the remote socket, which both consists of an IP address, and a port number. This gives us an identifier quadruple like this: (localaddress, localport, remoteaddress, remoteport). When the client

connects to a server, we will make sure that this connection is registered at the delaybox, and we also want to do this before the client sends the first SYN packet, so that the handshake is also delayed correctly. There are two ways to do this in OMNeT++: either we can call the *registerConnection()* method of the delaybox directly, or we can send a message directly to a special gate on the delaybox, so that the register connection message does not traverse the network and add to the traffic. We have chosen to simply register the connection by calling the *registerConnection()* method directly. This method takes both the identifier quadruple and the minimum RTT (as specified by the connection vector) as arguments and stores them in a list of connections. When registering a connection, the delaybox will also store the reverse connection for messages flowing the opposite way.

When a message arrives, the link-layer protocol will first handle the message, before passing the message to the delaybox. The delaybox functionality is quite simple. It reads the local address and remote address from the IP header, removes the IP header, and reads the local port and remote port from the TCP header. It tries to match this connection identifier against all the connections it has stored to find the minimum RTT of the connection. Once it finds a match, it will delay the message by half of the minimum RTT, before passing the message (with its IP header) to the network layer. When the message eventually is sent from the other end of the connection and back, the delaybox will match the reverse connection identifier with the same minimum RTT, and delay the message by half of the minimum RTT one more time, thus ensuring that no message will arrive at the sender with a smaller RTT than the minimum RTT.

Performance optimizing

For a while we had an issue with really slow running simulations. In some of the worst cases, we had simulations that had to run for weeks to finish, and this is unacceptable because it makes it impossible to properly test changes to the simulation script. The performance of the simulation seemed to drop the longer the simulation ran.

We had trouble for a while with closing connections properly without crashing the simulation. We circumvented this problem by just ignoring it, i.e., never closing any connections. The fact that each node and router never closes a connection should not have an impact on the simulation results, as hardware performance of nodes is not simulated in OMNeT++ anyway. As the list of connections is increasing, searching through the list of all connections becomes a menacing task, which is a problem.

We used Valgrind together with its tool Callgrind [3] to try to analyze our simulations and find bottlenecks in the code. We ran 30% of the dial-

[3]Valgrind[14] is an instrumentation framework for building dynamic analysis tools. There are many tools out there, for instance the memcheck tool which can be used to automatically detect memory-management problems. We have used the callgrind tool. This is a tool that records the call history among functions in a program's run. It collects information about the number of instructions executed, their relationship to source lines and so on. Kcachegrind is a program that is used to visualize callgrind output.

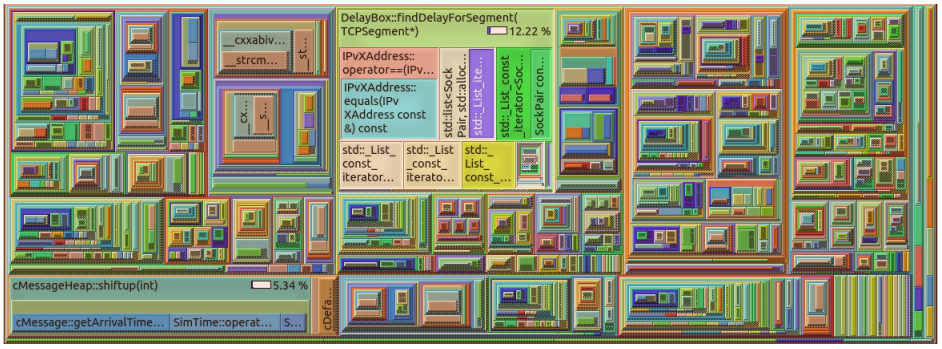


Figure 3.4: The callee map before optimization, shown in kCacheGrind

up basic scenario with valgrind to benchmark our simulation. Figure 3.4 shows the callee map from callgrind, shown in kCacheGrind. This map shows all the functions in the simulation, and the size represent how much of the simulation time has been spent in the function. In the middle of the map we can see a quite noticeable area highlighted. This is the DelayBox's *findDelayForSegment()* function, which is the function that looks through the connection list to find the minimum RTT of the connection. The rest of the callee map shows a lot of OMNeT++ internal functions. The program spent 12.22% of its time in the *findDelayForSegment()* function. This number in itself is not a problem, but it confirmed what we expected: that the DelayBox's lookup function was the main bottleneck. The number is quite small because we only ran the dial-up basic scenario for 30% of its time (running the simulation through Valgrind increases its run time significantly). This is the scenario with the least amount of traffic. We now knew that this was the main bottleneck, and we also knew that the list of connections was growing without limits, which as a really big problem for the other scenarios.

The first thought was that we could increase performance by decreasing the size of the list of connections in the DelayRouter. To do this, we had to figure out when connections could be safely closed without packets still being in flight. For a while we believed that the algorithm of the client and server was the problem, that we were actually closing connections at the incorrect time, but this was not the case. The problem was actually just a bug in the client. To close a connection we do three things: close the socket, remove the connection from the delaybox, and kill the client thread. When a client thread closes a socket, it means that the client is done sending ADUs and sends a FIN message. The server replies with its own FIN message. The problem was that when this FIN message arrived back at the main client thread, the main thread would still try to pass the FIN message to be the child thread, although this thread was already removed, thus causing a crash. The easy fix was to just let the main thread handle this FIN message on its own, rather than passing it to the child thread. We ran the dial-up scenario again for 30% of its time to compare our new simulation. The callee map is shown in Figure 3.5. As we can see, the highlighted part of the map is now noticeable smaller. The program now spends 9.36% of

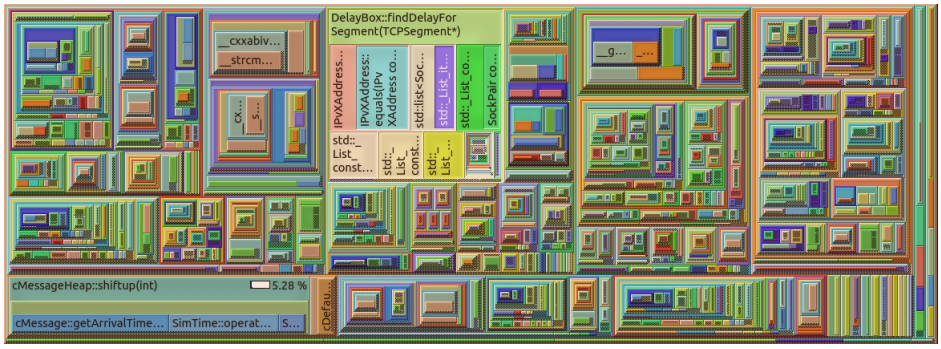


Figure 3.5: The callee map after first optimization, shown in kCacheGrind

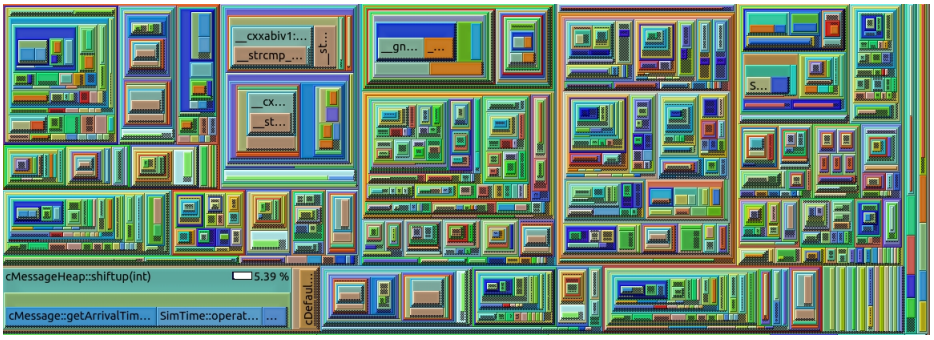


Figure 3.6: The callee map after storing connections in a hash map, shown in kCacheGrind

its time in the *findDelayForSegment()* function. This is a quite significant improvement, especially considering the fact that this bottleneck is a much bigger problem in the other test scenarios. It would be interesting to run a much larger simulation through valgrind as well, such as the access link scenarios for instance, but valgrind slows down the simulation to a crawl, and it would probably take months to finish it.

The performance of tmix has definitely been improved, but it is still a bottleneck in our simulations. The problem is that although the lookup time is shorter, the complexity of the lookup process is still *O(n)*. The idea that came to mind, is that the TCP module has the same amount of connections to look through, but it is still not a problem in terms of performance, so what does the TCP module do differently when looking up connections compared to our delaybox module? The answer is simple really. The TCP module does not have a list of connections at all, it has a hash map of connections, using the TCP identifier quadruple as a key. Hash maps have a complexity of *O(1)*, given that all keys map to exactly one value, which should be the case, as there should not exist two connections with the same identifier quadruple. After changing our implementation to store connections in a hash map as well, Figure 3.6 shows a callee map with no obvious bottlenecks.

Improving tmix

For a while the results we got from our simulations were quite off. The amount of traffic we saw in our OMNeT++ simulations were quite a lot lower than what we saw in the other two simulators. Something was obviously wrong. Thorough testing was required to find this bug. This revealed a couple of minor bugs that did not really have any impact on our simulations. What we did discover eventually was that there was quite a bit of traffic in the connection vector files that we never scheduled. The problem was that the connection vector file format used in the ns-2 version of the suite deviates a bit from the one described in the paper written by Weigle et al. [36], which we have used as a standard for our version of tmix, alongside the tmix description found in the ns-2 manual [27].

In the paper written by Weigle et al., a sequential connection is described in general as follows: *"a client makes series of k requests of sizes a1, a2, ..., ak, to a server that responds by transmitting k objects of sizes b1, b2, ..., bk, such that the i^{th} request is not made until the (i-1)st response is received in full."* As a result of this paper, we believed that in a sequential connection it was not possible for one of the communicating nodes to send several ADUs in a row. The connection vector files we have received however includes several sequential connections where this is the case.

The problem with this was that we had not really designed tmix to be able to handle such cases. Our design was that the initiator would send one ADU to the acceptor with instructions on how the acceptor would create its replying ADU. This was made in such a way that it was only possible to reply with one ADU. Handling several ADUs in a row from the initiator is easy. That is just a matter of checking whether the next ADU is from the initiator as well. If it is, then send the first one, and schedule the next ADU immediately. The problem is how to handle this in the acceptor.

Our solution is not perfect, but it should generate the correct amount of traffic. When the initiator detects that the acceptor is supposed to reply with several ADUs it will set a boolean saying that the acceptor is now handling a string of ADUs. While this boolean is set, the initiator will not send ADUs before it has received every ADU in the string. We have added three more attributes to the tmix message: *totalBytes*, *totalWaitTime*, and *numberOfAdus*. The initiator will count the aggregate amount of bytes, the aggregate amount of time to wait found the string of ADUs, as well as the total number of ADUs that the acceptor is supposed to send. This information is handed to the acceptor through the TmixMsg. Upon receiving this message, the acceptor knows how many ADUs it is supposed to send, the average amount of bytes for each ADU, and the average wait time between each ADU. To signal that the initiator should start sending ADUs again, the acceptor will use the *expectedReplyLength* field of the TmixMsg to mark the last ADU. The algorithm for creating these ADUs is straight forward and is not included here.

By handling this special case as we did, we still keep the acceptor quite simple, while the initiator is still in control. For these strings of ADUs, our version should send the correct amount of bytes over the correct amount of

time. However if there are differences in the wait time between each ADU, or if there are different sized ADUs described in the connection vector file, then our version of tmix will ignore this. All in all this should not have a significant impact on our simulation.

Shortcomings

There are a few shortcomings in our implementation of tmix for OMNeT++. The connection vectors specify a random loss rate. This loss rate is ignored in our version of tmix. This should not be a problem for our version of the evaluation suite in OMNeT++, as the ns-2 version of the suite does not use this loss rate anyway. This loss rate is not implemented in the ns-3 version of tmix either.

The connection vectors also specify an initial congestion window, but this is not implemented in our version either. TCP is designed in such a way that the initial congestion window starts really small, but grows quickly to find the capacity of the network. Because of the way that TCP is designed, you are always supposed to start with a small congestion window. The problem with the way that tmix is designed however, is that each connection vector does not necessary represent a connection from beginning to end, it may represent the latter half of a connection where the congestion window has already grown beyond the initial size. OMNeT++ does not take such connections into consideration, which means this is not easily done. Every TCP variant in OMNeT++ decides its own initial congestion window size. This is usually a multiplicative factor of the MSS. For instance for the NewReno version found in OMNeT++, the initial congestion window size is 4*MSS. There is no way to change this initial size without changing parts of the way that TCP has been implemented in OMNeT++. This shortcoming could have a significant impact on our simulations as the traffic for OMNeT++ should be a bit lower than for ns-2 and ns-3.

We have had quite a lot of problems with closing connection properly without crashing the simulation. In itself, it is not a problem having many connections up at the same time. We might get a slight performance hit, but it should not be a problem. The network should not act any differently whether there are many or few open connections. However when closing connections, each of the two nodes send a FIN message to each other, and this will affect the amount of traffic in the network, if only by a little. Instead of closing connections properly, we have sent a 0-sized ADU instead to simulate the FIN message. This is not optional, but it is better than nothing.

3.3.3 The evaluation suite implementation

The OMNeT++ TCP evaluation suite has been implemented according to the description in section 2.7. Where the NS2 version of the suite differs from the description of the suite, we have to chosen to make our version as close to the NS2 version as possible, to make the results comparable.

The simulation suite is bundled together in a folder named eval, and the placement of this folder is optional. For further instructions on how to use the OMNeT++ evaluation suite, see Appendix B.

Architecture

The test suite consists of a few components. There is a NED file named dumbbell.NED, which creates the basic dumbbell topology. This includes the creation of six TmixHosts, a DelayRouter and a normal Router, as well as the connections between these components.

Most of the suite is found in the omnetpp.ini file, which includes all the configurations for all the different scenarios. Each configuration consists of values that determine for instance the bandwidth of each link, propagation delay of each link, which connection vector files to use, and the size of each queue.

The topology file and the configuration file together is enough to run the simulation. In addition to the simulation, there is a file named dumbbell.anf. *.anf* denotes an analysis file in OMNeT++. This file specifies what shall be finally displayed by filtering the simulation results. Results are written in comma-separated values (CSV) format. The CSV file will be run through a very simple R script to calculate the final results.

Topology and protocol stack

The simulation setup is simpler for the OMNeT++ version of the evaluation suite, than it is for ns-2 and ns-3. The reason for this is that the INET framework contains several modules that are very useful for our purpose, such as the StandardNode module and the Router module. The StandardNode module is a module representing a typical node, with a full TCP/IP stack, ready to use for Internet scenarios. It includes normal link-layer protocols, such as PPP, Ethernet and wireless LAN. It includes both IPv4 and IPv6. The StandardHost contains TCP, UDP and SCTP, as well as applications on top of these transport-layer protocols. The Router module contains just the three lower levels of the protocol-stack, which is what we need. The StandardNode has been extended to be used with Tmix, and is known as a TmixHost, and we have also extended the normal router to be used with tmix, and is known as a DelayRouter (as described in more detail in section 3.3.2.

The basic setup for most of our tests is described in a file named dumbbell.ned, which describes a basic dumbbell topology, shown in Figure 3.7. The figure shows six nodes, numbered one through six, all of which are TmixHosts. There are two kinds of routers shown in the figure, the one on the right-hand side is a normal Router module, as found in the INET framework, while the router on the left-hand side is a DelayRouter, which is an extended Router used in Tmix.

The setup is much like the ns-3 setup. The links are "perfect" (no bit errors) so nothing is really required of the link-layer protocol. We use the point-to-point protocol in the link layer. The queues used are

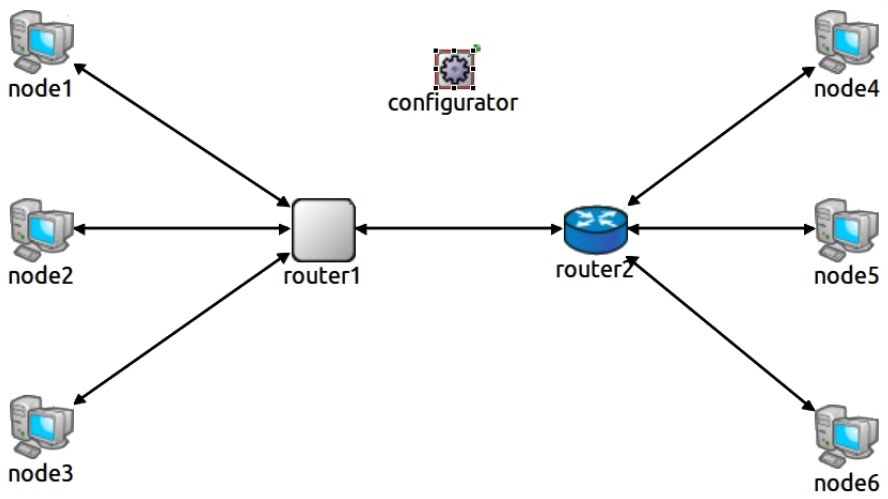


Figure 3.7: Dumbbell topology shown in OMNeT++

DropTailQueues, and the queue sizes used are the same as for ns-3. The network-layer protocol we use is IPv4, as there is no reason (and it should not have a significant impact on the simulation anyway) to use IPv6.

There are not that many TCP variants implemented directly in OMNeT++ (when we say OMNeT++ in this case we mean both the basic OMNeT++ simulator, as well as the INET framework), but with the Network Simulation Cradle (NSC) installed, it is possible to run real world TCP/IP protocol stacks inside of OMNeT++. NSC comes with four network stacks: Linux, FreeBSD, OpenBSD and the lightweight TCP/IP stack (lwIP) [7]. NSC in OMNeT++ however has its limitations. First of all it only supports the Linux stack and the lwIP stack, and secondly the support for each of them is quite limited. It would have been interesting to run the Linux stack with CUBIC, as we done in ns-2 and in ns-3, but unfortunately this is not possible as NSC in OMNeT++ is today. One of the problems with NSC in OMNeT++ is the fact that it is not possible to change the maximum segment size (MSS) for the Linux stack. The maximum transmission unit (MTU) is always set to 1500 bytes, thus making the MSS 1460 bytes. This is not compatible with our traffic generator. Another problem is that the Linux stack does not support the object transfer mode, which is the one that tmix uses. These two limitations make it impossible for us to run NSC with the Linux stack together with tmix.

The BSD stacks are not supported at all in NSC for OMNeT++. That leaves us with the last stack, the lwIP stack. The lwIP stack is meant to be used in small embedded systems with small amounts of RAM and ROM. It does not make sense to use such a stack in our scenarios as tmix models a range of different users, some of which do not fit the lwIP stack very well. The TCP variant used in lwIP is much like NewReno.

Since NSC for OMNeT++ is not useable for us, we have chosen to use the three built-in TCP variants in OMNeT++: Tahoe, Reno, and NewReno.

Traffic generation

To generate traffic we have used our own version of tmix implemented for OMNeT++, described in detail in section 3.3.2. Traffic is described through a series of connection vector files, given as input to the simulation through the configuration file. There are nine files in total, where one file describes the traffic between a pair of nodes in the simulation. Each host in the simulation contains a TmixSrv that the other clients can connect to. Each node from the left side connects to every node from the right side and vice versa. All traffic flows through the central link, which means that the nodes on either the left side, or the right, do not communicate among themselves. The setup is the same as in ns-2 and in ns-3.

As mentioned in the section about tmix for OMNeT++, tmix has been extended so that we have the possibility of scaling traffic. The scales for each scenario are the same as for the NS2 version of the evaluation suite.

Collecting and presenting statistics

There are a number of statistics to be collected, as described in the evaluation suite. As mentioned, OMNeT++ has a built-in system to handle statistics, and many of the modules in the INET framework have several very useful pre-defined statistics, that we use to collect our statistics. Note that for every simulation, it is possible to define a warm-up period. In this warm-up statistics are not collected. This warm-up period is set according to the warm-up periods defined in the evaluation suite.

To get the throughput of the central link, there is a statistic named *rxPkOk:sum(packetBytes)* found in the PPP module, which is the sum of all bytes successfully received. We can use this statistic from both the routers to calculate the throughput in each direction. Throughput is here defined as the number of bits per second traversing the central link, so calculating the throughput is just a matter of dividing the number of bytes by the number of seconds that the simulation ran for. This definition of throughput also makes it easy to derive the link utilization: throughput divided on the link capacity.

To find the drop rate in each direction, there is a statistic named *dropPk:count* that is generated by the queue of each router. In addition to the drop count in each router, there is also a statistic named *rxPkOk:count* found in the PPP module of each router, which counts the number of packets that were successfully received. These two statistics together makes it possible to calculate the drop rate.

A histogram is generated for each queue, which shows the queuing time of each packet. There are also several scalars based on this histogram. These scalars include the aggregate mean queuing delay through the entire simulation, and the aggregate mean queuing size.

To generate statistics, we have used the analysis tool in the IDE to create a dataset consisting of the values described above. This dataset is written as a CSV file, and further run through a basic R script to create the final statistics. The R script is very simple: it takes the link capacity as argument,

reads the CSV file from OMNeT++, and calculates the final statistics based on these values.

shortcomings

Our version of tmix skips a few details from the connection vector files. Loss rate is ignored (just like ns-2 and ns-3 does), but also the initial congestion window size is ignored, which makes our OMNeT++ version of tmix different to the ones found in ns-2 and ns-3.

As a whole the evaluation suite for OMNeT++ is not very easy to use. Now this is not a big problem really as it will not be released for public use as the TCP evaluation suite suffers from a number of problems. A main script to automatically perform every task required to produce the results should have been in place. The main problem with this is that we have used the IDE to develop the suite, and to analyse our results using the analysis tool that came with the IDE. This analysis tool does not accept command line commands, which makes it difficult to create such a script. OMNeT++ does however include an analysis tool named *scavetool* that is command line based. We have not prioritized rewriting the analysis part of the suite however, so for now the suite will have to be run in several steps.

Chapter 4

Simulation results

4.1 Introduction

An important question to answer is: *"what are we actually looking for?"*. The TCP evaluation suite (described in Section 2.7) defines a number of test scenarios, and a number of metrics that are collected during the course of the simulation. There are two general approaches to our analysis:

First of all, there are several studies (which are listed below in the actual analysis) that compare and evaluate existing TCP variants. These studies should give us an indication on how TCP is expected to behave under certain conditions. To analyse our results we can use these studies to try and draw some conclusion on the simulated performance of each TCP variant in the different simulators.

In addition to comparing the results from each simulator with studies on TCP, we will also analyse our results by comparing the results between the simulators to try and see whether they differ in behaviour.

There are several metrics that we can evaluate. Metrics that are very important when evaluating TCP is throughput, delay, and drop rate. There will always be a tradeoff between the three. Sending packets more aggressively will lead to a higher throughput, but also to a higher delay as well as a higher drop rate.

In this case we define *throughput* to be the amount of raw data transferred over the network per time unit. This also includes all control packets, such as ACKs being sent back and forth.

Delay measures the amount of delay a packet suffers when travelling through a network. The amount of delay for a single packet increases in several steps along the path. It takes an amount of time to push a packet onto a link (based on the link's bandwidth). It takes a while for the packet to propagate each link, based on the medium used and the length the link. Whenever the packet reaches a router it might have to wait in queue if the network is congested etc. Round-trip time (RTT) is the cumulative delay for a packet being sent from the sender to the receiver, and back again. While RTT definitely gives an indication on how the network performs, queuing delay (which is the amount of time each packet waits in queue) ignores the network topology and user think times entirely, thus giving a

better indication of the congestion level of the network.

Drop rate is the number of packets dropped compared to the number of packets that were successfully delivered. Whether a packet drop occurs because of congestion (which causes a queue to drop the packet), because of bit-errors on the link (which may cause the link-layer protocol to drop the packet), or because of any other problem that might cause the packet to be lost, does not make a difference to TCP. In our simulations, the only type of drop we should see is the one caused by congestion as other details (such as link-layer errors) are abstracted away.

4.2 The expected results

First of all we expect the different TCP variants to behave relative to each other as seen in other research. In this section we will go through a few research papers that compare the performance of TCP variants to see what is expected.

Floyd and Fall [19] explore the benefits of adding selective acknowledgements (SACK) to TCP. They have a simple simulation setup where they send 20 packets and increment the loss rate for every run to see how each variant reacts to an increased amount of loss. On the first run, one packet is lost, on the second run, two packets are lost etc. This is done four times. The experiment is run on the original *ns* simulator. Reno gives optimal performance when a single packet is dropped. In the presence of a single drop, Reno fast retransmits the packet before going into fast recovery. The problem is that whenever several drops occur in rapid succession, Reno lowers *ssthresh* several times, which lowers the throughput severely. In the scenarios with three or four lost packets, Reno will have to wait for a retransmit timer to recover. NewReno and Reno with SACK enabled look quite alike each other. However, the NewReno sender is at most able to retransmit one packet every round-trip time (RTT). They also show that when there are several lost packets from the same window, Tahoe significantly outperforms Reno, and barely outperforms NewReno. Both Reno and NewReno outperform Tahoe when there are few lost packets from the same window. They compare the results to a Reno trace taken from actual Internet traffic measurements. They conclude that the trace exhibits behaviour that is similar to the one seen in simulation.

Bateman et al. [18] compares the performance of several high speed TCP variants in ns-2 and in a real-world test bed using Linux. Although we do not evaluate high speed TCP variants (such as BIC and CUBIC) in this thesis because of lack of support, it is interesting to see whether ns-2 results are comparable to the real world. They have a simple setup with low complexity (a dumbbell topology) that is meant to highlight the behaviour of the TCP variants, rather than being realistic. They report that the ns-2 simulations in general behave very much like the test-bed does. They believe that ns-2 simulations can be used to report behaviour that reflects use of TCP variants in operational use. They use the same code in both the test-bed and in ns-2, and encourage more possibilities for inclusion of

real code in ns-2. Although they conclude that the ns-2 results and testbed results to be equivalent, they also say that as complexity increases (more complex testbed, additional factors such as CPU scheduling hardware variations etc.) we may see more variation in the results. All in all they conclude that ns-2 is a valuable tool to investigate protocols, although there is always a small uncertainty to whether the results are actually realistic.

Jansen and McGregor [24] present their validation of the Network Simulation Cradle (NSC) for ns-2. They compare the results from a testbed network to a simulation using NSC. Both the simulation and the testbed run the same protocol stack (which is the point of NSC after all). They conclude that the accuracy in results are very good, and that it is worthwhile to use real world based TCP stacks.

We have a good idea on how TCP should behave, but how do we expect the network simulators to compare to each other? The TCP variants are clearly defined in RFCs, so in theory (given that all the optional options are configured identically across the network simulators) they should perform very similar to each other given that the network and traffic generator are equal in all three simulators. We do not expect the results to be exactly the same across the simulators - that is unrealistic - but we do expect them to give similar results as we model exactly the same scenario in all three.

4.3 ns-2 results

The results from the ns-2 basic scenarios are summarized in Table 4.1 and Table 4.2. Each scenario was run for four TCP variants: Tahoe, Reno, NewReno, and Cubic. For each TCP variant, each scenario was run three times with three levels of congestion: no congestion, mild congestion, and moderate congestion. For each run, the throughput, total number of packets, total number of dropped packets, average queuing delay, average queue size, and loss rate was collected and are presented in the tables. There are five basic scenarios run (the wireless basic scenario is not run as they are not implemented in ns-3 and OMNeT++): Access Link (AL), Data Center (DC), Geostationary Satellite (GS), Trans-Oceanic link (TO), and Dial-Up (DU). The level of traffic is much higher in the reverse direction (from right to left), than it is in the normal direction (left to right), except for in the geostationary satellite connection where the central link is asymmetric. As the reverse direction has the highest amount of traffic, this is also the direction we will focus on.

CUBIC is a TCP variant that is designed to utilize network resources by quickly finding the maximum network capacity. This means that in most cases, CUBIC should have a very good throughput, although at the expense of having a higher drop rate. This is seen very clearly in every ns-2 test. CUBIC shows the highest throughput in both directions in almost every single scenario. CUBIC is very aggressive (at least compared to the other three variants), this is seen very clearly as well, as CUBIC has the highest drop rate in most scenarios, and especially in the reverse direction. Not surprisingly, on the other end of the scale, we see that Tahoe has the

Scenario	Throughput (Mbps)		Packets		Drops		Queueing delay (ms)		Queue size (B)		Loss rate (%)	
	L->R	R->L	L->R	R->L	L->R	R->L	L->R	R->L	L->R	R->L	L->R	R->L
AL - Tahoe - uncon.	11.135995	51.855756	1252090	1844237	0	0	0.0106	0.311	17	2.1e+03	0.000	0.000
AL - Reno - uncon.	11.135803	51.914825	1253393	1845741	0	0	0.0105	0.317	16.6	2.14e+03	0.000	0.000
AL - NewReno - uncon.	11.144948	52.164500	1256412	1852138	0	0	0.0108	0.317	17.6	2.16e+03	0.000	0.000
AL - Cubic - uncon.	11.318910	52.215007	1257521	1851346	0	0	0.0121	0.349	20.1	2.38e+03	0.000	0.000
AL - Tahoe - mild.	22.054923	99.029588	2436266	3504344	0	15913	0.0259	40.7	85.9	5.12e+05	0.000	0.454
AL - Reno - mild.	22.069437	99.269682	2437203	3509288	0	13202	0.0264	41.5	88.3	5.22e+05	0.000	0.376
AL - NewReno - mild.	22.082636	99.080388	2436096	3503487	0	17056	0.0259	41.7	85.5	5.24e+05	0.000	0.487
AL - Cubic - mild.	22.677841	99.866564	2515419	3562617	0	42441	0.0285	49.4	95.3	6.23e+05	0.000	1.191
AL - Tahoe - mod.	23.081433	99.999639	2562748	3598361	0	59650	0.0275	60.8	93.8	7.64e+05	0.000	1.658
AL - Reno - mod.	23.271498	99.998393	2556634	3588610	0	50781	0.0273	59.5	91.9	7.47e+05	0.000	1.415
AL - NewReno - mod.	23.185062	99.999670	2553123	3589445	0	54265	0.0267	60.2	88.3	7.56e+05	0.000	1.512
AL - Cubic - mod.	24.019087	99.999665	2694802	3669822	0	131969	0.030	65.8	104	8.25e+05	0.000	3.596
DC - Tahoe - uncon.	97.652810	518.047473	2166193	3320996	0	0	0.00116	0.0322	22.3	2.1e+03	0.000	0.000
DC - Reno - uncon.	99.750395	511.491722	2160138	3294437	0	0	0.0012	0.0300	22.9	1.95e+03	0.000	0.000
DC - NewReno - uncon.	97.874643	511.467448	2151859	3290091	0	0	0.00114	0.0303	20.9	1.97e+03	0.000	0.000
DC - Cubic - uncon.	99.461384	516.709266	2140206	3275735	0	0	0.00110	0.0298	18	1.97e+03	0.000	0.000
DC - Tahoe - mild.	205.449661	993.336775	4308987	6400501	40556	19635	0.0793	7.73	4.91e+03	9.64e+05	0.941	0.307
DC - Reno - mild.	205.061808	994.791068	4312873	6414585	39166	18546	0.0836	8.58	5.17e+03	1.07e+06	0.908	0.289
DC - NewReno - mild.	203.764636	998.749544	4332772	6457876	39982	23805	0.0797	10.3	4.91e+03	1.29e+06	0.923	0.369
DC - Cubic - mild.	212.487884	999.971124	4427422	6504589	17732	121487	0.0905	14.1	4.82e+03	1.77e+06	0.401	1.868
DC - Tahoe - mod.	215.310107	998.566678	4513354	6580725	44493	92589	0.0849	13.9	5.69e+03	1.74e+06	0.986	1.407
DC - Reno - mod.	216.622035	998.674840	4487529	6545928	43883	72642	0.0814	13.7	5.32e+03	1.72e+06	0.978	1.110
DC - NewReno - mod.	215.408483	998.700629	4488702	6545369	45233	88818	0.0826	13.9	5.48e+03	1.75e+06	1.008	1.357
DC - Cubic - mod.	227.701084	999.981958	4655892	6627218	19577	262490	0.102	15.2	5.54e+03	1.91e+06	0.420	3.961
GS - Tahoe - uncon.	2.136908	8.703579	547936	757009	0	0	4.44	0.496	1.48e+03	621	0.000	0.000
GS - Reno - uncon.	2.136908	8.703579	547936	757009	0	0	4.44	0.496	1.48e+03	621	0.000	0.000
GS - NewReno - uncon.	2.132607	8.604784	545151	753597	0	0	4.42	0.459	1.48e+03	567	0.000	0.000
GS - Cubic - uncon.	2.374769	9.689081	641577	883586	0	0	6.55	0.57	2.47e+03	802	0.000	0.000
GS - Tahoe - mild.	3.801869	16.329494	1017394	1408142	12674	0	60.30	1.06	2.96e+04	2.4e+03	1.246	0.000
GS - Reno - mild.	3.818484	16.292358	1027132	1430278	18549	0	67.20	1.07	3.28e+04	2.38e+03	1.806	0.000
GS - NewReno - mild.	3.799282	16.319469	1027367	1430019	15738	0	62.10	1.09	3.04e+04	2.41e+03	1.532	0.000
GS - Cubic - mild.	3.988752	17.870001	1233321	1595291	172270	0	131.00	1.09	6.58e+04	2.74e+03	13.968	0.000

Table 4.1: Results from the basic scenarios for ns-2. AL = Access Link, DC = Data Center, TO = Trans-Oceanic link, GS = Geostationary Satellite, DU = Dial-Up link

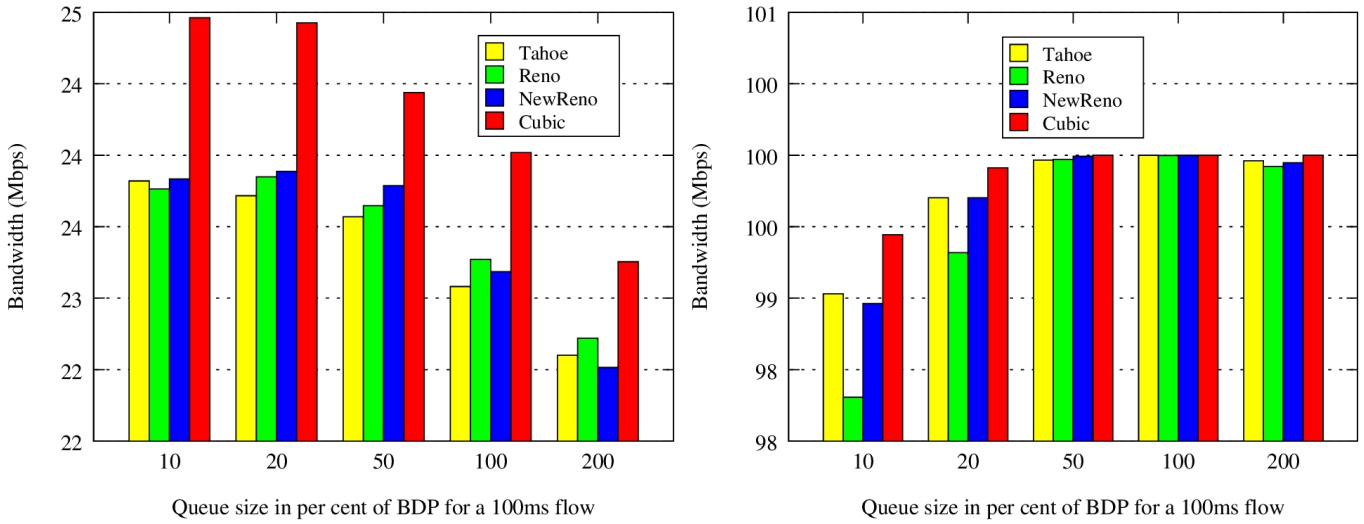
Scenario	Throughput (Mbps)		Packets		Drops		Queueing delay (ms)		Queue size (B)		Loss rate (%)	
	L->R	R->L	L->R	R->L	L->R	R->L	L->R	R->L	L->R	R->L	L->R	R->L
GS - Tahoe - mod.	3.900370	16.868534	1078853	1486739	51601	0	93.3	1.17	4.58e+04	2.72e+03	4.783	0.000
GS - Reno - mod.	3.857608	17.377933	1087883	1506822	50597	0	87.3	1.23	4.26e+04	2.86e+03	4.651	0.000
GS - NewReno - mod.	3.861167	17.068640	1082870	1487079	61432	0	94.6	1.2	4.62e+04	2.83e+03	5.673	0.000
GS - Cubic - mod.	3.997646	18.322520	1307784	1644327	257865	0	142.0	1.15	7.11e+04	2.91e+03	19.718	0.000
TO - Tahoe - uncon.	143.218754	606.075778	3181110	4514823	0	0	0.00134	0.0334	25.5	2.59e+03	0.000	0.000
TO - Reno - uncon.	143.314689	607.619164	3185692	4523718	0	0	0.00138	0.0334	26.8	2.58e+03	0.000	0.000
TO - NewReno - uncon.	143.090220	604.013048	3175257	4504457	0	0	0.00137	0.0328	26.7	2.54e+03	0.000	0.000
TO - Cubic - uncon.	149.211451	629.856305	3297234	4661817	0	0	0.00157	0.0433	31.6	3.54e+03	0.000	0.000
TO - Tahoe - mild.	290.569761	999.983519	5771619	7649589	0	30595	0.00429	143.0	190	1.79e+07	0.000	0.400
TO - Reno - mild.	290.543068	999.983338	5751812	7638440	0	22992	0.00425	141.0	188	1.76e+07	0.000	0.301
TO - NewReno - mild.	290.463255	999.983224	5770805	7655876	0	31314	0.00430	144.0	191	1.8e+07	0.000	0.409
TO - Cubic - mild.	308.940536	999.983369	6290937	7985289	0	220288	0.00461	148.0	206	1.86e+07	0.000	2.759
TO - Tahoe - mod.	310.473124	999.983303	6059719	7814043	0	86533	0.00454	146.0	206	1.82e+07	0.000	1.107
TO - Reno - mod.	310.052233	999.983381	6049032	7800292	0	80794	0.00468	146.0	216	1.83e+07	0.000	1.036
TO - NewReno - mod.	307.174513	999.983242	6067113	7812054	0	106036	0.00462	147.0	211	1.84e+07	0.000	1.357
TO - Cubic - mod.	332.279036	999.983429	6678118	8258384	0	410913	0.00513	147.0	244	1.84e+07	0.000	4.976
DU - Tahoe - uncon.	0.004167	0.015221	38418	46963	114	1628	4.33	254.0	97.8	650	0.297	3.467
DU - Reno - uncon.	0.004178	0.014998	38101	46473	103	1650	4.34	265.0	97.5	671	0.270	3.550
DU - NewReno - uncon.	0.004171	0.015097	38132	46537	114	1627	4.38	268.0	97.5	679	0.299	3.496
DU - Cubic - uncon.	0.004548	0.015734	56370	64794	409	3883	3.65	241.0	105	781	0.726	5.993
DU - Tahoe - mild.	0.006393	0.025481	63104	75615	172	3876	3.55	278.0	107	1.13e+03	0.273	5.126
DU - Reno - mild.	0.006320	0.022681	57045	67185	157	3460	3.85	279.0	107	1.03e+03	0.275	5.150
DU - NewReno - mild.	0.006330	0.022635	56908	67049	181	3563	3.89	289.0	107	1.06e+03	0.318	5.314
DU - Cubic - mild.	0.007157	0.024296	94831	107737	574	8953	2.96	227.0	110	1.09e+03	0.605	8.310
DU - Tahoe - mod.	0.013556	0.056739	135496	161359	866	24559	3.71	403.0	125	3.07e+03	0.639	15.220
DU - Reno - mod.	0.013704	0.051752	132885	164124	852	25029	3.81	357.0	126	2.58e+03	0.641	15.250
DU - NewReno - mod.	0.013645	0.052359	132239	160897	889	23025	3.82	362.0	130	2.63e+03	0.672	14.310
DU - Cubic - mod.	0.015602	0.052308	234087	273475	3285	61936	2.96	247.0	103	1.86e+03	1.403	22.648

Table 4.2: Results from the basic scenarios for ns-2. AL = Access Link, DC = Data Center, TO = Trans-Oceanic link, GS = Geostationary Satellite, DU = Dial-Up link

lowest throughput in most scenarios. While Cubic is a very clear winner in throughput in almost every test, there is not that much difference between the other three variants. NewReno consistently performs better than Reno and Tahoe in terms of throughput, while Reno consistently performs better than Tahoe. This is what we would expect of the four TCP variants. There are a few scenarios where Tahoe outperforms Reno. For instance in the uncongested data center scenario, Tahoe has a higher throughput in the reverse direction than both Reno and NewReno. This is not normal, but it is not unheard of either. One could think that Tahoe outperform Reno and NewReno in the long run when running uncongested scenarios due to a higher complexity in the algorithms of Reno and NewReno, but this is abstracted away in simulations, so this is not the case. What strikes us as unusual however is that Reno outperforms NewReno in some scenarios. For instance, the throughput for Reno is higher than NewReno in the uncongested data center, in the uncongested geostationary satellite scenario, and also in the trans-oceanic link scenario. As far as we know, there is no reason why Reno should never outperform NewReno in terms of throughput.

As we know, there is a tradeoff between throughput, loss rate, and queuing delay. To get a high throughput, there always has to be packets available in the central queue, but a larger queue leads to a longer wait time, while a full queue leads to packet drops. This means that the rest of the numbers are secondary in interest as the throughput sort of gives the loss rate and queuing delay of each variant. Higher throughput means higher delay and higher loss rate. The variants that performed best in terms of throughput performed the worst in terms of queuing delay and loss rate. This is mostly seen in the results. There are a few scenarios where this does not happen, for instance in the moderately congested data center scenario, Tahoe is the worst in terms of throughput, but not the best in terms of queuing delay, yet the numbers are so close to each other that we cannot conclude with anything other than the fact that these differences may just be happenstance.

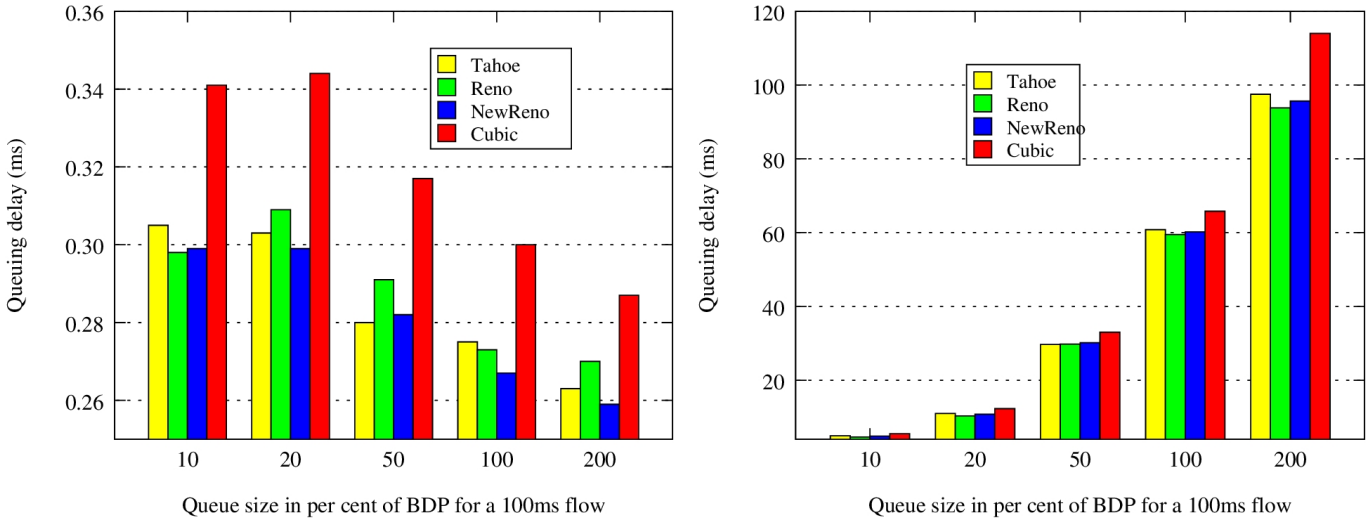
The next scenario is the delay-throughput tradeoff scenario. This scenario investigates the different TCP variants as the queue size changes. The queue size is given as a per cent of the bandwidth-delay-product (BDP) for a 100 ms flow. Figure 4.1a and Figure 4.1b shows the throughput of each test as a function of queue size in the normal direction, and in the reverse direction, respectively. Again, we can see that CUBIC outperforms the other variants by quite a bit when it comes to throughput. The results are very much like one would expect. With a longer queue there are room for more packets, and as long as there are packets to send, throughput is close to the link capacity. What looks a bit strange though is that in the normal direction, a larger queue gives a lower throughput. One would think that if the queue sizes are increased, then there would be more unused capacity if the small queue was large enough to handle a certain throughput. This might be explained by packets being dropped in the reverse direction however. When the queue size is smaller, the drop rate is higher. When segments are sent in the normal direction, but ACKs are lost in the reverse



(a) Normal direction

(b) Reverse direction

Figure 4.1: Throughput shown as a function of queue size - the delay-throughput tradeoff scenario in ns-2



(a) Normal direction

(b) Reverse direction

Figure 4.2: Queueing delay shown as a function of queue size - the delay-throughput tradeoff scenario in ns-2

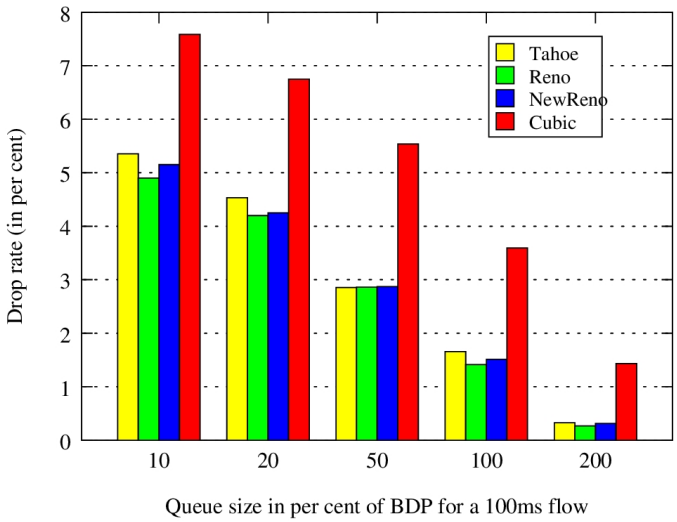


Figure 4.3: Drop rate shown as a function of queue size - the delay-throughput tradeoff scenario in ns-2

direction, this may cause the sender to retransmit segments. This adds to the average throughput of the link in the normal direction. Because of this we actually see a lower throughout in the tests with the largest queues as there are fewer retransmissions.

As mentioned above, Tahoe is better at handling bursts of packet loss than Reno and NewReno without SACK. When the queue size is really small, the chance of this happening is much greater; therefore Tahoe outperforms Reno and NewReno in these cases.

Figure 4.2a and Figure 4.2b show the queuing delay for each test as a function of queue size in the normal direction, and in the reverse direction, respectively. Again, we can see the throughput-delay tradeoff clearly by comparing these two graphs to the throughput graphs. A longer queue gives the possibility for longer queuing delay, this is obvious. We can also see that the delay of CUBIC is always longer than the delay for the other variants. This is because of the aggressive nature of CUBIC, always filling up the queues more than the other variants do. In the normal direction we see a lower queuing delay for the larger queues. This is explained by a higher congestion in the reverse direction, as explained above.

Figure 4.3 shows the drop rate of each test as a function of queue size in the reverse direction. The normal direction is left out because the drop rate is always close to or exactly zero, which is not very interesting. The drop rate is much the other way around. A large queue stores more packets for a longer amount of time (as seen in the previous two graphs) instead of dropping packets. Again, CUBIC, and its aggressive increase in congestion window, has a far higher drop rate than any of the other variants. The other three variants are more or less equal in terms of drop rate.

The conclusion is that a larger queue gives a better throughput as there always are packets available to send. A large queue also gives a smaller loss rate as the queue can handle more packets at the same time. However,

the larger the queue is, the longer the queuing delay gets. This is a tradeoff that one has to choose between. These results are very much like one would expect to see. Tahoe, Reno and NewReno perform quite close to each other, while CUBIC is the only one that really stands out in these results. This is expected as NewReno is based on Reno, and Reno is based on Tahoe. They all share the same fundamental algorithm. CUBIC on the other hand has a very different growth function, and therefore acts very differently.

While we mostly see results that one would expect to see, there are still a few things in these results that we cannot explain. For instance, why does Reno outperform NewReno in some scenarios? This makes little sense, and should be investigated further.

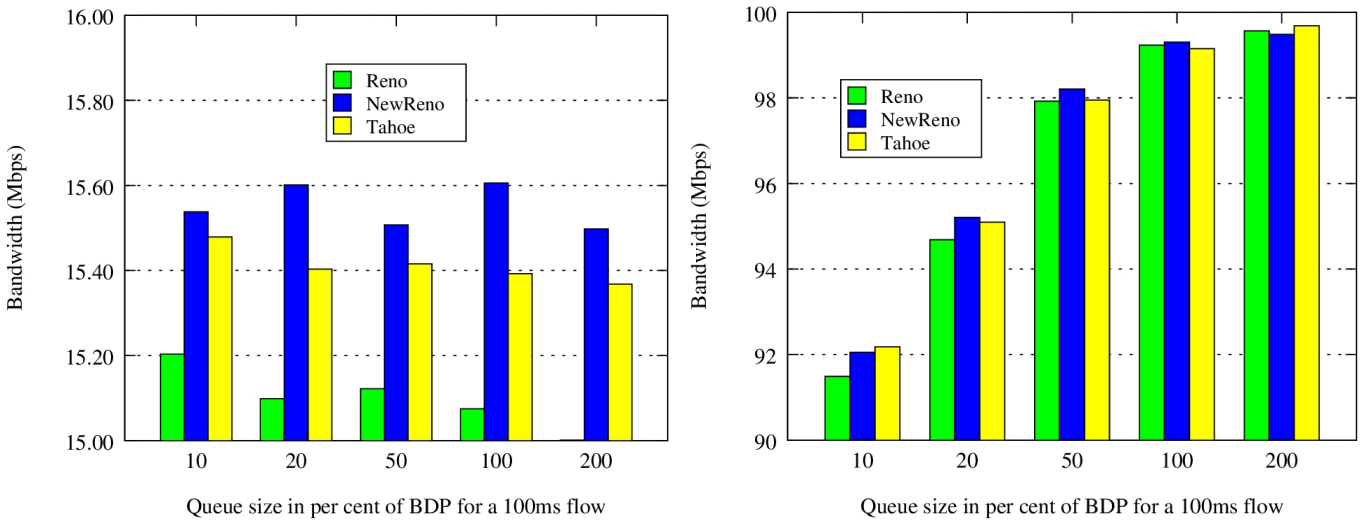
4.4 ns-3 results

We have a good grasp on how each TCP variant is expected to behave, and we know the results from ns-2. Given that both simulators works perfectly, and given that both test suites are implemented correctly, then these results should be fairly similar. In this section we will go through the ns-3 results without comparing them to the results from ns-2 however. The simulation setup is as close as possible to the setup in ns-2. There are only four basic scenarios unfortunately, compared to the five in ns-2. The trans-oceanic link scenario is missing. As we do not close sockets properly in tmix for ns-3, this scenario crashes after running for a very long time. We have been unable to work around this problem. This problem also applied to the data center scenario. We have decreased the total run time of this scenario down to 30 seconds, where the warm-up period is 5 seconds. Each TCP variant in each basic scenario is run three times with three levels of congestion: uncongested, mild congestion and moderate congestion. In ns-3 we evaluate only Tahoe, Reno and NewReno. Table 4.3 summarizes the results of all the basic scenarios. For each basic scenario we present the average throughput, the total amount of packets, the total amount of dropped packets, the average queuing delay, the average loss rate and the average link utilization in both directions.

In almost every single scenario, NewReno outperforms both Tahoe and Reno in terms of throughput. This is what we expected. In some scenarios however, for instance in the uncongested and mildly congested access link scenario, we can see that in the reverse direction (which is where the highest amount of traffic occurs) Tahoe is the winner when it comes to throughput. Again, the fact that Tahoe outperforms Reno and NewReno is not unheard of, but it is quite rare. The results from these scenarios are the aggregate results from running a scenario over time where the nature of the traffic constantly changes. It is weird, yet not impossible, that the average results over time from Tahoe should outperform Reno and NewReno. In ns-3, NewReno always outperforms Reno, except for in the moderately congested dial-up scenario. As mentioned earlier, Tahoe performs better than Reno and NewReno whenever several packets from the same congestion window are lost. The moderately congested dial-up

Scenario	Throughput (Mbps)		Total packets		Dropped packets		Avg. queuing delay(ms)		Loss rate (%)		Link utilization (%)	
	L->R	R->L	L->R	R->L	L->R	R->L	L->R	R->L	L->R	R->L	L->R	R->L
AL - Tahoe - uncon.	7.814282	55.793153	1517289	2161875	0	0	1.0	2.1	0.0000	0.0000	7.81	55.79
AL - Reno - uncon.	7.677453	55.071420	1493473	2136192	0	0	1.0	2.1	0.0000	0.0000	7.68	55.07
AL - NewReno - uncon.	7.893047	55.213328	1507043	2147178	0	0	1.0	2.1	0.0000	0.0000	7.89	55.21
AL - Tahoe - mild.	14752610	98565090	2853972	3920750	0	61605	1.0	38.2	0.0000	1.5783	14.75	98.57
AL - Reno - mild.	14.402825	98.163373	2824821	3900945	0	59987	1.0	37.6	0.0000	1.5444	14.40	98.16
AL - NewReno - mild.	14.952926	98.403510	2870795	3931605	0	72294	1.037	39.1	0.0000	1.8480	14.95	98.40
AL - Tahoe - mod.	53.92578	99.152054	2980320	4034565	0	111790	1.0	44.3	0.0000	2.7833	15.39	99.15
AL - Reno - mod.	50.74953	99.233188	2956658	4023199	0	104652	1.0	44.3	0.0000	2.6130	15.07	99.23
AL - NewReno - mod.	56.05566	99.303097	3009784	4061044	0	128348	1.0	46.2	0.0000	3.1763	15.61	99.30
DC - Tahoe - uncon.	63.460124	405.175174	1002991	1376856	0	0	0.002	0.073	0.0000	0.0000	6.35	40.52
DC - Reno - uncon.	63.065892	400.805141	994314	1365315	0	0	0.000	0.066	0.0000	0.0000	6.31	40.08
DC - NewReno - uncon.	63.117286	401.742747	995825	1367097	0	0	0.002	0.068	0.0000	0.0000	6.31	40.17
DC - Tahoe - mild.	121.155671	817.795727	1981425	2741065	0	551	0.003	1.469	0.0000	0.0202	12.12	81.78
DC - Reno - mild.	121.058374	815.507528	1975218	2737797	0	287	0.003	1.314	0.0000	0.0105	12.11	81.55
DC - NewReno - mild.	120.800840	818.171213	1981010	2749266	0	184	0.003	1.382	0.0000	0.0067	12.08	81.82
DC - Tahoe - mod.	148.763189	979.484653	2459194	3312382	0	15194	0.003	6.503	0.0000	0.4610	14.88	97.95
DC - Reno - mod.	149.232020	973.243804	2440684	3286236	0	14703	0.004	6.399	0.0000	0.4497	14.92	97.32
GS - Tahoe - uncon.	1.902554	14.318922	936701	1328939	1576	0	15.94	151.8	0.1740	0.0000	47.56	35.80
GS - Reno - uncon.	1.867241	14.381934	927032	1328510	1225	0	15.86	152.1	0.1367	0.0000	46.68	35.95
GS - NewReno - uncon.	1.957917	14.546937	946362	1349201	2194	0	16.05	152.2	0.2397	0.0000	48.95	36.37
GS - Tahoe - mild.	3.325787	27.793039	1802176	2569923	39576	0	17.31	157	2.2694	0.0000	83.14	69.48
GS - Reno - mild.	3.330291	27.787008	1792652	2574592	38090	0	17.27	158.3	2.1962	0.0000	83.26	69.47
GS - NewReno - mild.	3.417792	27.897739	1815330	2595194	58432	0	17.81	158.7	3.3250	0.0000	85.44	69.74
GS - Tahoe - mod.	3.479821	28.864112	1887512	2678064	54425	0	17.79	158.6	2.9810	0.0000	87.00	72.16
GS - Reno - mod.	3.495089	28.941278	1885827	2694493	59791	0	17.87	159.9	3.2778	0.0000	87.38	72.35
GS - NewReno - mod.	3.543279	29.018968	1894137	2690328	77350	0	18.29	160.1	4.2202	0.0000	88.58	72.55
DU - Tahoe - uncon.	0.002842	0.016089	38088	56957	300	6148	52.7	365.0	0.8094	10.8239	4.44	25.14
DU - Reno - uncon.	0.002847	0.016210	38867	59831	265	8731	51.6	365.9	0.7002	14.6317	4.45	25.33
DU - NewReno - uncon.	0.002866	0.016391	40682	61273	313	9391	47.6	381.1	0.7892	15.3679	4.48	25.61
DU - Tahoe - mild.	0.004040	0.027192	63706	95400	359	12382	40.5	375.8	0.5820	13.0230	6.31	42.49
DU - Reno - mild.	0.004063	0.027342	65309	100379	387	16905	39.4	375.9	0.6115	16.8954	6.35	42.72
DU - NewReno - mild.	0.004087	0.027560	67209	102319	369	18163	38.0	390.4	0.5661	17.8091	6.39	43.06
DU - Tahoe - mod.	0.009151	0.061548	157281	245825	1510	67556	39.6	490.3	0.9917	27.5638	14.30	96.17
DU - Reno - mod.	0.009246	0.061377	159739	254074	1691	76778	39.9	497.6	1.0929	30.3064	14.45	95.90
DU - NewReno - mod.	0.009230	0.060864	159348	251442	1550	74992	40.0	492.7	1.0044	29.9129	14.42	95.10

Table 4.3: Results from the basic scenarios for ns-3. AL = Access Link, DC = Data Center, GS = Geostationary Satellite, DU = Dial-Up link



(a) Normal direction

(b) Reverse direction

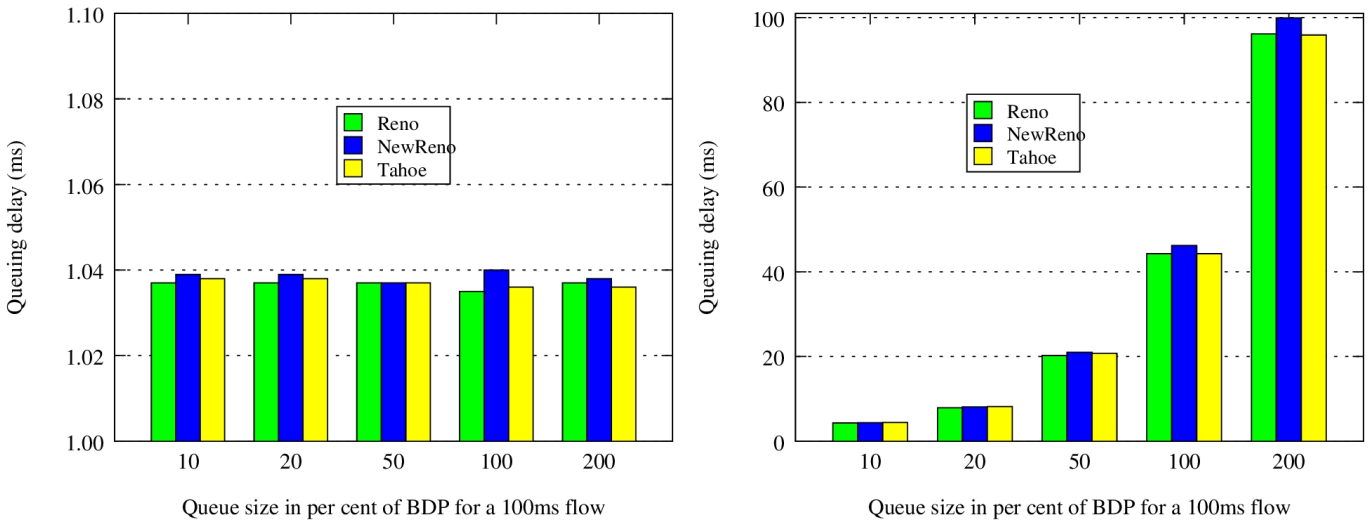
Figure 4.4: Throughput shown as a function of queue size - the delay-throughput tradeoff scenario in ns-3

scenario has the highest drop rate of all the tests. This may explain why Tahoe outperforms the others in this case.

The other results, loss rate and queuing delay for instance, are not that interesting, as they are more or less given when we know the throughput. We can see that NewReno has the highest amount of queuing delay in most scenarios, but that is to be expected as NewReno populates the queues in a greater sense than Tahoe and Reno does. Where this is not the case, the results are usually so close to each other that we may conclude that they occurred because of chance.

The delay-throughput tradeoff scenario investigates how the different TCP variants behave as the queue size varies. The queue size is given as a per cent of the bandwidth-delay-product (BDP) for a 100 ms flow. The bandwidth in the normal direction and reverse direction respectively is shown in Figure 4.4a and 4.4b. In the reverse direction the results are very much like what we expect. A higher queue gives a higher throughput as there always are packets available for transfer. NewReno performs the best in most cases, although it has the worst performance for the largest queue size. Again, the fact that NewReno is outperformed by Reno is unexpected. It must be noted that the three variants perform very close to each other so that no significant conclusions may be drawn. Queuing delay and drop rate is also very much like expected. A higher queue gives a higher queuing delay, as well as a lower drop rate. NewReno, the most aggressive of the three variants, has both higher queuing delay and a higher drop rate in all tests.

The results for the normal direction seem to be more or less unaffected by queue sizes. We can see that the throughput is generally very low compared to the throughput seen in the reverse direction, only at about 15% of the link capacity. This is also the reason why the queue size does



(a) Normal direction

(b) Reverse direction

Figure 4.5: Queueing delay shown as a function of queue size - the delay-throughput tradeoff scenario in ns-3

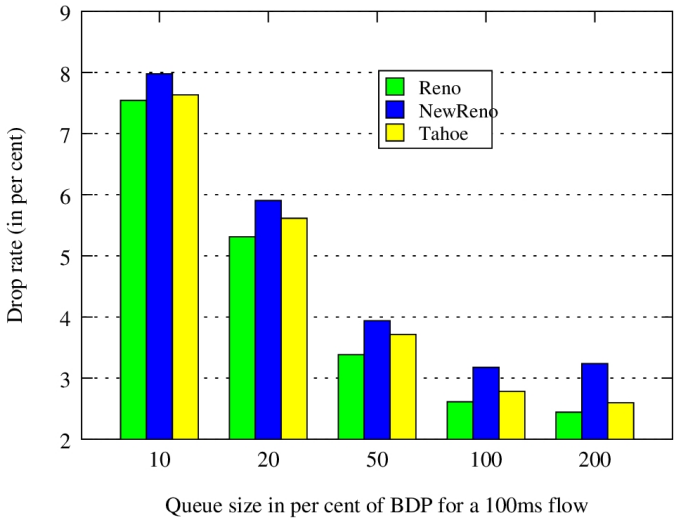


Figure 4.6: Drop rate shown as a function of queue size - the delay-throughput tradeoff scenario in ns-3

not really affect the results, because there is always capacity to handle what little traffic is flowing in this direction no matter how small the queue is. If the smallest queue is enough to handle the traffic, then a larger queue will not really make any difference.

All in all the TCP variants seem to behave according to what we would expect in ns-3.

4.5 OMNeT++ results

The setup for OMNeT++ is the same as for the other scenarios. Each scenario is run three times with three levels of congestion: uncongested, mild congestion, and moderate congestion. We test three TCP variants in OMNeT++: Tahoe, Reno and NewReno. We run the same five basic scenarios as in ns-2. The results of these basic scenarios are summarized in Table 4.4 and Table 4.5.

We can see very clearly that these results do not correspond to the expected results. The three TCP variants all perform more or less exactly the same. This is seen very clearly in the link utilization columns, where the link utilization percentage is rounded. Because it is rounded, the link utilization is exactly the same in almost every test for the three variants. This should not be the case.

The question we have to try to answer is why these results are so close to each other. The first thing that came to mind was that we actually failed to switch TCP algorithm, but this does not seem to be the case. There is no randomness involved in these tests, which means that if the same test is run twice, then the results will be exactly the same. Because of this, it is clear that *something* happens when we change TCP algorithm. Then there is the question of whether we change the algorithm for only one node, but this should not be the case either. There is only one line in the configuration file that decides the TCP algorithm to use, which looks like the following for Reno, NewReno and Tahoe respectively:

```
**.tcp.tcpAlgorithmClass = "TCPReno"
**.tcp.tcpAlgorithmClass = "TCPNewReno"
**.tcp.tcpAlgorithmClass = "TCPTahoe"
```

Only one of these is present in the configuration file for each scenario run, so that there is definitely no problem with precedence of commands. The wildcard notation '**' denotes that every single tcp module in the simulation should use the given tcpAlgorithmClass. We are therefore confident that we are in fact running the correct TCP algorithm.

Next we try to consider whether there are scenarios where the three TCP variants *should* behave similar to each other. They all share a common slow start and congestion avoidance phase. The three variants handle packet drops and retransmissions differently from each other, but if there are no packet drops, then they should behave exactly the same. In the intuitive sense however, TCP is "greedy", which means that if a TCP sender has data to send, it will keep increasing its sending rate until a packet

Scenario	Throughput (Mbps)		Drops		Queueing delay (ms)		Queue size (B)		Loss rate (%)		Link utilization (%)	
	L->R	R->L	L->R	R->L	L->R	R->L	L->R	R->L	L->R	R->L	L->R	R->L
AL - Tahoe - uncon.	8.275600	45.321333	0	0	0.0041164	0.1	0.0279	0.7226	0	0	8.28	45.32
AL - Reno - uncon.	8.275573	45.321600	0	0	0.0041180	0.1	0.0280	0.7227	0	0	8.28	45.32
AL - NewReno - uncon.	8.275600	45.321867	0	0	0.0041191	0.1	0.0280	0.7213	0	0	8.28	45.32
AL - Tahoe - mild.	15.867387	90.031467	0	754	0.0084427	15.6	0.1129	202.107	0	0.0194	15.87	90.03
AL - Reno - mild.	15.866187	90.031467	0	769	0.0086502	15.6	0.1157	202.005	0	0.0198	15.87	90.03
AL - NewReno - mild.	15.867387	90.030133	0	682	0.0084743	15.7	0.1134	204.203	0	0.0175	15.87	90.03
AL - Tahoe - mod.	16.607067	93.705600	0	11900	0.0090552	22.9	0.1260	308.82	0	0.2938	16.61	93.71
AL - Reno - mod.	16.606213	93.705600	0	12919	0.0090360	23.1	0.1258	310.874	0	0.3188	16.61	93.71
AL - NewReno - mod.	16.614560	93.702933	0	15674	0.0090169	23.3	0.1257	314.579	0	0.3858	16.61	93.70
DC - Tahoe - uncon.	67.701382	392.106182	0	0	0.00031034	0.0079001	0.0183	0.4504	0	0	6.77	39.21
DC - Reno - uncon.	67.701382	392.106182	0	0	0.00031034	0.0079001	0.0183	0.4504	0	0	6.77	39.21
DC - NewReno - uncon.	67.701382	392.106182	0	0	0.00031034	0.0079001	0.0183	0.4504	0	0	6.77	39.21
DC - Tahoe - mild.	130.423127	796.545455	0	0	0.00060298	0.1	0.0715	11.8433	0	0	13.04	79.65
DC - Reno - mild.	130.423127	796.545455	0	0	0.00060298	0.1	0.0715	11.8433	0	0	13.04	79.65
DC - NewReno - mild.	130.423127	796.545455	0	0	0.00060298	0.1	0.0715	11.8433	0	0	13.04	79.65
DC - Tahoe - mod.	138.332509	845.048727	0	0	0.00065471	0.2	0.0824	23.7263	0	0	13.83	84.50
DC - Reno - mod.	138.332509	845.048727	0	0	0.00065471	0.2	0.0824	23.7263	0	0	13.83	84.50
DC - NewReno - mod.	138.332509	845.048727	0	0	0.00065471	0.2	0.0824	23.7263	0	0	13.83	84.50

Table 4.4: Results from the basic scenarios for OMNeT++. AL = Access Link, DC = Data Center, GS = Geostationary Satellite, DU = Dial-Up link

Scenario	Throughput (Mbps)		Drops		Queueing delay (ms)		Queue size (B)		Loss rate (%)		Link utilization (%)	
	L->R	R->L	L->R	R->L	L->R	R->L	L->R	R->L	L->R	R->L	L->R	R->L
GS - Tahoe - uncon.	1.748823	8.570503	0	0	1.3	0.2	1.8437	0.2600	0	0	43.72	21.43
GS - Reno - uncon.	1.749166	8.571440	0	0	1.3	0.2	1.8210	0.2656	0	0	43.73	21.43
GS - NewReno - uncon.	1.748880	8.571829	0	0	1.3	0.2	1.8338	0.2643	0	0	43.72	21.43
GS - Tahoe - mild.	3.396709	16.740229	0	0	47.5	0.5	126.3	1.3435	0	0	84.92	41.85
GS - Reno - mild.	3.397714	16.745829	0	0	47.4	0.5	126.156	1.3521	0	0	84.94	41.86
GS - NewReno - mild.	3.397863	16.755886	0	0	47.4	0.5	126.113	1.3517	0	0	84.95	41.89
GS - Tahoe - mod.	3.550114	17.308800	0	0	114.4	0.6	317.25	1.4876	0	0	88.75	43.27
GS - Reno - mod.	3.550023	17.314400	0	0	114.3	0.6	316.876	1.4702	0	0	88.75	43.29
GS - NewReno - mod.	3.550629	17.328000	0	0	114.5	0.5	317.618	1.4658	0	0	88.77	43.32
DU - Tahoe - uncon.	0.003374	0.015309	5	1648	14.6	284.0	0.0472	0.8989	0.0078	2.5405	5.27	23.92
DU - Reno - uncon.	0.003375	0.015325	5	1780	14.6	293.2	0.0472	0.9289	0.0077	2.7457	5.27	23.95
DU - NewReno - uncon.	0.003370	0.015273	5	1885	14.8	299.7	0.0477	0.9459	0.0078	2.9026	5.27	23.86
DU - Tahoe - mild.	0.004676	0.025376	17	3152	10.7	317.9	0.0501	1.4541	0.0181	3.3308	7.31	39.65
DU - Reno - mild.	0.004680	0.025407	19	3344	10.6	323.5	0.0500	1.4807	0.0202	3.5245	7.31	39.70
DU - NewReno - mild.	0.004673	0.025316	17	3572	10.8	330.6	0.0505	1.5062	0.0182	3.7721	7.30	39.56
DU - Tahoe - mod.	0.010054	0.060052	429	44572	17.7	508.2	0.1720	4.6927	0.2201	19.4423	15.71	93.83
DU - Reno - mod.	0.010099	0.060226	430	46116	17.7	511.5	0.1733	4.7430	0.2196	19.9147	15.78	94.10
DU - NewReno - mod.	0.010076	0.060089	473	45231	18.2	512.2	0.1767	4.7247	0.2428	19.6911	15.74	93.89

Table 4.5: Results from the basic scenarios for OMNeT++. AL = Access Link, DC = Data Center, GS = Geostationary Satellite, DU = Dial-Up link

Scenario	uncongested	mild congestion	moderate congestion
Data center	0.488	0.244	0.222
Access link	5.2	2.6	2.4
Trans-oceanic link	0.242	0.121	0.11
Geostationary satellite	20	10	9
Dial-up link	24800	12400	47
Delay/throughput	5.2	2.6	2.4

Table 4.6: New scales in OMNeT++

drop occurs. This means that even if the amount of traffic on the network is relatively low, TCP will increase its sending rate until the network is congested. BUT! This is given that the TCP sender actually has data to send. If the traffic only consists of a lot of small data transfers, then it is in theory possible to have a large simulation with many flows and quite high link utilization, without a single packet drop.

There definitely exist scenarios where the three TCP variants behave similar to each other, but these scenarios are highly unrealistic. Small non-greedy flows are very common in the Internet according to Weigle et al. [36]. These flows are typically seen in HTTP requests where a client asks for a web page, and the server returns it. There is no time for TCP to ramp up its sending rate in this case because there is only a single round of transmissions.

As we can see, the total amount of transmitted bytes in OMNeT++ is quite a bit lower than for ns-2 and for ns-3. Because of this, the drop rate is very low compared to the other simulators as well. The different TCP variants handle packet loss differently, but should behave very similarly if there no packets are lost. Our idea was that the amount of transmitted bytes in OMNeT++ was too low, and that this could possibly explain the results we got. To explore this, we ran all the tests for OMNeT++ once more, but with a lower tmix scale to increase the amount of traffic. The new tmix scales are seen in Table 4.6, and the new results are seen in Table 4.7 and Table 4.8. Note that we have included the trans-oceanic link scenario in these new results. We have the same problem with closing connections properly in OMNeT++ as in ns-3. This made the simulation crash after running for a while. As a workaround we have decreased the run time of this scenario.

Figure 4.7a and Figure 4.7b shows the bandwidth in the delay-throughput tradeoff scenario in the normal direction, and in the reverse direction, respectively. Figure 4.9 shows the loss rate in the reverse direction. We can see that when the queue size is lower, the loss rate is higher (as expected), and we can also see that the difference between the three TCP variants are much greater when the loss rate is higher. Figure 4.5a and Figure 4.5b show the queuing delay in each direction. All these results are much like we would expect: a large queue gives a higher throughput, higher queuing delay, and a higher loss rate.

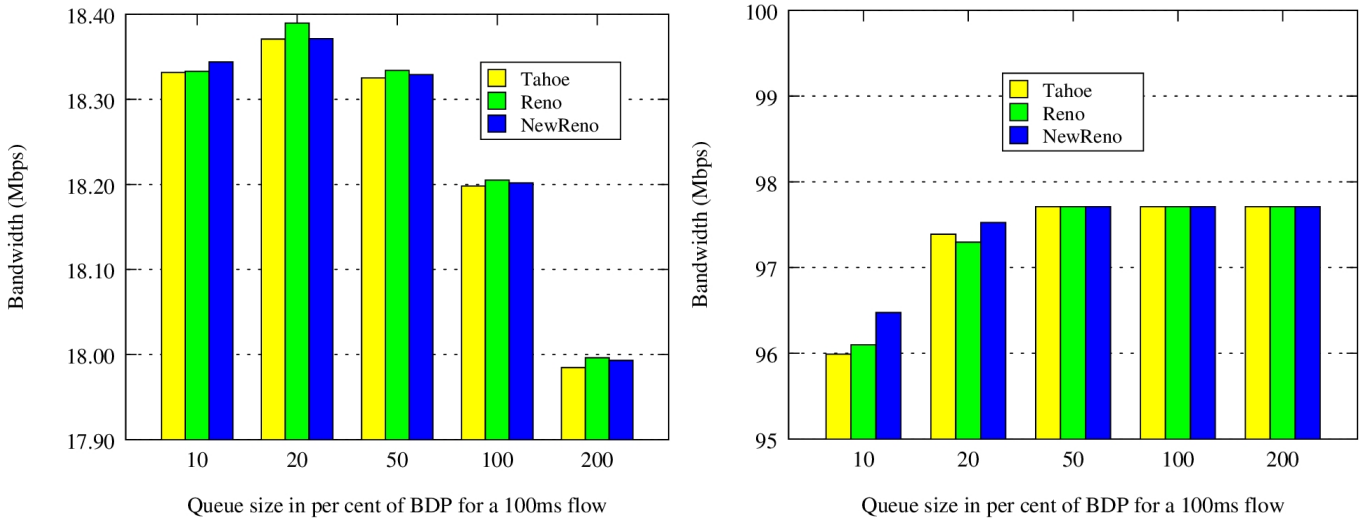
What we have seen is that the three variants differ as loss rate increases.

Scenario	Throughput (Mbps)		Drops		Queueing delay (ms)		Queue size (B)		Loss rate (%)		Link utilization (%)	
	L->R	R->L	L->R	R->L	L->R	R->L	L->R	R->L	L->R	R->L	L->R	R->L
AL - Tahoe - uncon.	9.470667	51.808267	0	0	0.0047586	0.1	0.0371	1.1132	0	0	9.47	51.81
AL - Reno - uncon.	9.470667	51.808267	0	0	0.0047554	0.1	0.0370	1.1152	0	0	9.47	51.81
AL - NewReno - uncon.	9.470480	51.808533	0	0	0.0047527	0.1	0.0370	1.1133	0	0	9.47	51.81
AL - Tahoe - mild.	17.299973	95.962667	0	36931	0.0096155	30.8	0.1374	425.482	0	0.8847	17.30	95.96
AL - Reno - mild.	17.304987	95.962667	0	38159	0.0095261	30.9	0.1363	426.456	0	0.9124	17.30	95.96
AL - NewReno - mild.	17.314427	95.962667	0	48249	0.0096026	31.1	0.1376	429.983	0	1.1499	17.31	95.96
AL - Tahoe - mod.	18.198133	97.710133	0	89920	0.010538	39.8	0.1547	561.023	0	2.0800	18.20	97.71
AL - Reno - mod.	18.205067	97.710133	0	92672	0.010490	39.5	0.1543	558.899	0	2.1390	18.21	97.71
AL - NewReno - mod.	18.201893	97.710133	0	111023	0.010536	40.0	0.1553	566.847	0	2.5484	18.20	97.71
DC - Tahoe - uncon.	81.861236	486.964364	0	0	0.00036594	0.012316	0.0268	0.8698	0	0	8.19	48.70
DC - Reno - uncon.	81.861236	486.964364	0	0	0.00036594	0.012316	0.02686	0.8698	0	0	8.19	48.70
DC - NewReno - uncon.	81.861236	486.964364	0	0	0.00036594	0.012316	0.0268	0.8698	0	0	8.19	48.70
DC - Tahoe - mild.	159.194182	943.421091	0	10126	0.00079353	4.9	0.1128	676.167	0	0.1342	15.92	94.34
DC - Reno - mild.	159.201455	943.421091	0	10656	0.00079622	4.9	0.1132	677.952	0	0.1411	15.92	94.34
DC - NewReno - mild.	159.175273	943.421091	0	17252	0.00079396	5.1	0.1131	696.405	0	0.2278	15.9175	94.3421
DC - Tahoe - mod.	175.613091	978.434910	0	52702	0.0010113	7.0	0.1503	993.166	0	0.6663	17.56	97.84
DC - Reno - mod.	175.584000	978.434909	0	54577	0.0010291	7.0	0.1530	998.265	0	0.6898	17.56	97.84
DC - NewReno - mod.	175.136000	978.434909	0	82188	0.00099133	7.1	0.1480	1014.13	0	1.0310	17.51	97.84
TO - Tahoe - uncon.	107.547840	490.448000	0	0	0.00052422	0.015037	0.0455	1.2316	0	0	10.75	49.04
TO - Reno - uncon.	107.547840	490.448000	0	0	0.00052422	0.015037	0.0455	1.2316	0	0	10.75	49.04
TO - NewReno - uncon.	107.547840	490.448000	0	0	0.00052422	0.015037	0.0455	1.2316	0	0	10.75	49.04
TO - Tahoe - mild.	204.162560	929.440000	0	0	0.0010751	19.2	0.1799	3060.46	0	0	20.42	92.94
TO - Reno - mild.	204.162560	929.440000	0	0	0.0010751	19.2	0.1799	3060.46	0	0	20.42	92.94
TO - NewReno - mild.	204.162560	929.440000	0	0	0.0010751	19.2	0.1799	3060.46	0	0	20.42	92.94
TO - Tahoe - mod.	218.128640	964.966400	0	17799	0.0011785	49.3	0.2071	8144.8	0	0.4315	21.81	96.50
TO - Reno - mod.	218.128640	964.966400	0	17799	0.011785	49.3	0.2071	8144.8	0	0.4315	21.81	96.50
TO - NewReno - mod.	218.128640	964.966400	0	17799	0.0011785	49.3	0.2071	8144.8	0	0.4315	21.81	96.50

Table 4.7: Results from the newly scaled basic scenarios for OMNeT++. AL = Access Link, DC = Data Center, TO = Trans-Oceanic link, GS = Geostationary Satellite, DU = Dial-Up link

Scenario	Throughput (Mbps)		Drops		Queueing delay (ms)		Queue size (B)		Loss rate (%)		Link utilization (%)	
	L->R	R->L	L->R	R->L	L->R	R->L	L->R	R->L	L->R	R->L	L->R	R->L
GS - Tahoe - uncon.	2.220994	10.902754	0	0	2.3	0.3	3.9474	0.4511	0	0	55.52	27.26
GS - Reno - uncon.	2.221097	10.902651	0	0	2.3	0.3	3.9609	0.4499	0	0	55.53	27.26
GS - NewReno - uncon.	2.221429	10.908754	0	0	2.2	0.3	3.8859	0.4531	0	0	55.54	27.27
GS - Tahoe - mild.	3.832606	18.778857	62764	0	435.3	0.7	1298.84	2.0069	2.9194	0	95.82	46.95
GS - Reno - mild.	3.832789	18.756457	63009	0	437.0	0.7	1303	1.9815	2.9334	0	95.82	46.89
GS - NewReno - mild.	3.832857	18.753829	66960	0	436.8	0.7	1301.3	2.0147	3.1142	0	95.82	46.88
GS - Tahoe - mod.	3.911977	19.837029	200092	0	549.9	0.8	1663.13	2.5697	8.6422	0	97.80	49.59
GS - Reno - mod.	3.911977	19.672800	212667	0	558.2	0.8	1670.01	2.4799	9.2264	0	97.80	49.18
GS - NewReno - mod.	3.912046	19.721829	213199	0	556.3	0.8	1671.54	2.5123	9.2110	0	97.80	49.30
DU - Tahoe - uncon.	0.003408	0.015442	5	1721	13.8	284.5	0.0448	0.9060	0.0077	2.6336	5.33	24.13
DU - Reno - uncon.	0.003411	0.015444	5	1731	13.8	292.6	0.0449	0.9321	0.0077	2.6574	5.33	24.13
DU - NewReno - uncon.	0.003404	0.015399	5	1908	14.0	299.1	0.0454	0.9494	0.0077	2.9205	5.32	24.06
DU - Tahoe - mild.	0.004744	0.025827	24	3279	10.6	317.8	0.0508	1.479	0.0251	3.4030	7.41	40.35
DU - Reno - mild.	0.004744	0.025848	25	3369	10.6	322.9	0.0509	1.5040	0.0261	3.4900	7.41	40.39
DU - NewReno - mild.	0.004736	0.025771	24	3632	10.8	329.7	0.0513	1.5301	0.0252	3.7650	7.40	40.27
DU - Tahoe - mod.	0.010169	0.060664	450	47873	17.8	516.3	0.1751	4.8013	0.2287	20.4697	15.89	94.79
DU - Reno - mod.	0.010172	0.060698	535	47328	18.0	513.3	0.1775	4.7929	0.2711	20.2199	15.89	94.84
DU - NewReno - mod.	0.010182	0.060604	465	46628	17.9	513.5	0.1760	4.7891	0.2358	19.9986	15.91	94.69

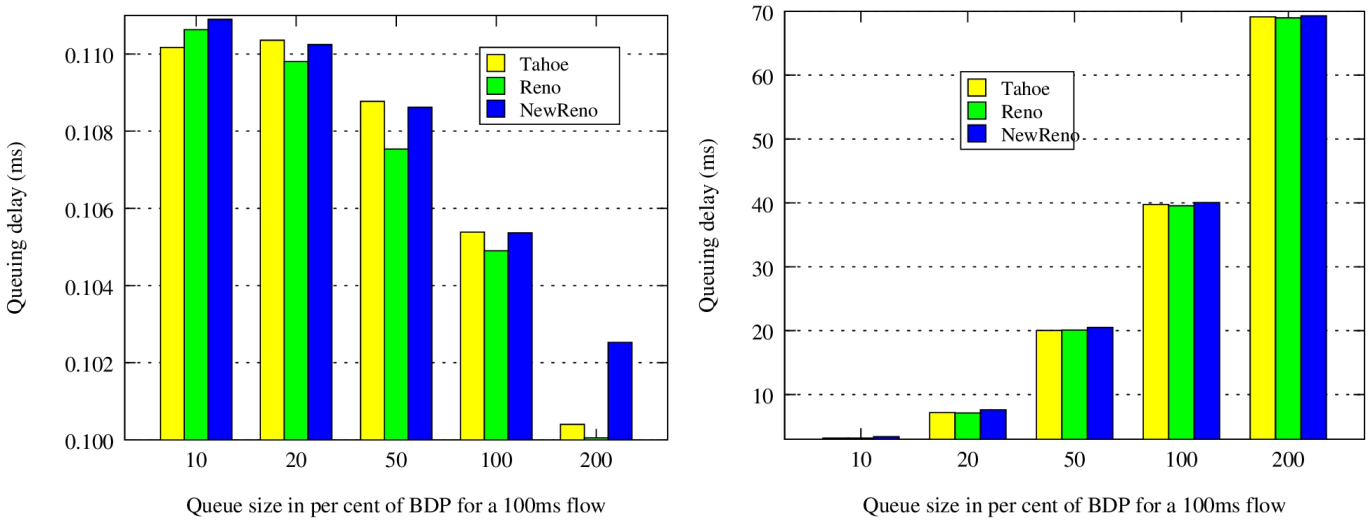
Table 4.8: Results from the newly scaled basic scenarios for OMNeT++. AL = Access Link, DC = Data Center, TO = Trans-Oceanic link, GS = Geostationary Satellite, DU = Dial-Up link



(a) Normal direction

(b) Reverse direction

Figure 4.7: Throughput shown as a function of queue size - the delay-throughput tradeoff scenario in OMNeT++



(a) Normal direction

(b) Reverse direction

Figure 4.8: Queueing delay shown as a function of queue size - the delay-throughput tradeoff scenario in OMNeT++

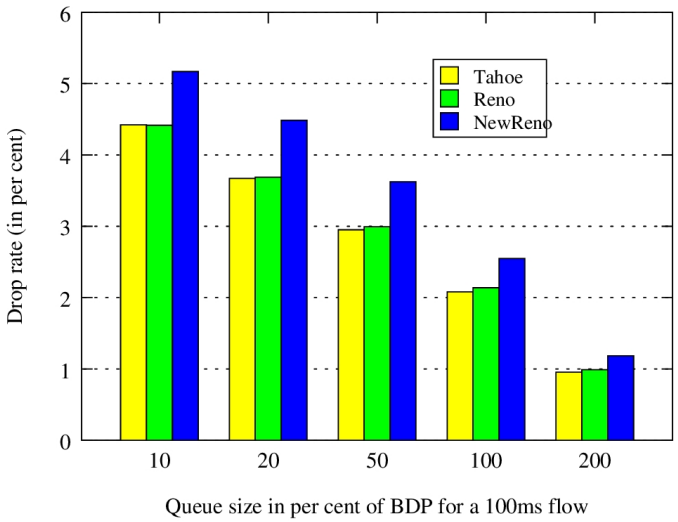


Figure 4.9: Drop rate shown as a function of queue size - the delay-throughput tradeoff scenario in OMNeT++

The newly scaled tests show a higher loss rate in each scenario, but still the results from each variant are very close to each other. We have to investigate the nature of the traffic as well. We have checked that the traffic generator generates the correct type of traffic in a number of different scenarios. We have created test files with only one ADU to see whether the traffic generator could handle different types of ADUs. These tests seem to generate the correct results. However, is it possible that at some point in the connection vector files, there is a special case that our traffic generator does not properly handle, thus ruining the traffic for the rest of the simulation? We can see that the results from the tests with the largest amounts of traffic seem to be closer to each other than the results from the tests with the smaller amounts of traffic. This indicates that this problem is in fact possible.

If every single connection consists of just a single request and a single response (like typically seen in HTTP applications), then there should hardly be any difference between the TCP variants (as they do not have time to ramp up their sending rate properly). We have a run a simple reference test to see how the variants behave in the presence of greedy flows to see how the variants react. (TODO: Lag test).

4.6 Comparison

We have run a common set of tests using three different network simulators. We have seen in ns-2 and ns-3 that the evaluated TCP variants seem to behave the way they should. In OMNeT++ we see some very strange results where the three tested TCP variants behave very similar to each other.

It is not possible to compare the results across the network simulators

however, as the numbers are quite far off. If the evaluation suite was modelled perfectly in all three simulators, we could draw the conclusion that at least two of three network simulators do not function as they should. However the problem is that there is quite a bit of uncertainty in a number of things, and especially in the traffic generator, so that we cannot draw such a conclusion. In this section we consider a few details in which we have been uncertain and discuss the significance of these details.

For each pair of communicating initiator/acceptor pair in the simulation, there is a connection vector file describing the traffic. We have tried combining different files with different node-pairs, but this has not changed the results significantly. We are confident that we now use the same connection vector file in each of the three simulators to describe the traffic between the same initiator/acceptor pair.

There are quite a few options in the configuration of TCP. We have mostly used the default values unless otherwise described in the evaluation suite. Delayed acks, selective acknowledgements (SACK), window scaling, and timestamp support is disabled by default. Nagle's algorithm is enabled for ns-2 and OMNeT++, but is not supported in ns-3. These options should be mostly equal for every simulator. We also know that OMNeT++ is different from ns-2 and ns-3 because the OMNeT++ version of tmix ignores the initial congestion window size specified in the connection vector files. To see the effect of setting the correct initial congestion window size, we have run a few of the scenarios in ns-3 without this functionality to see how it affects the results. We have run the dial-up basic scenario with all congestion levels, and we have also run the access link uncongested. The results are seen in Table 4.9 and we have included parts of the original results in Table 4.10 as well so that they are easier to compare. What we can see in these results is that the new results have a slightly lower throughput, and the new results are also slightly closer to each other than the original results. These numbers are, however, not like the results we see in OMNeT++, so we can conclude that the initial congestion window size functionality in itself is not the reason for why the OMNeT++ results are strange.

The most likely culprit in this case is the traffic generator. There is a large difference between the results from the different simulators. This can probably not be explained by small differences such as TCP options and such. A difference in how the traffic generator works may have the sort of impact that we see in our results.

4.7 Discussion

In the beginning of this thesis we were focused on trying to verify, or at least explore, the realism of network simulators. We chose the TCP evaluation suite as a basis for our test cases. As the thesis has progressed, the focus has shifted more and more from the verification of network simulators, to working on the TCP evaluation suite, and as a consequence, the tmix traffic generator.

Congestion level	Throughput (bps)		Avg. queuing delay		Loss rate (%)	
	L->R	R->L	L->R	R->L	L->R	R->L
DU - Tahoe - uncon.	2809	15585	0.052390	0.342037	0.269842	2.781614
DU - Reno - uncon.	2794	15531	0.052616	0.337858	0.260358	2.517607
DU - NewReno - uncon.	2780	15498	0.053475	0.343087	0.292814	2.768426
DU - Tahoe - mild.	4005	26317	0.038714	0.353536	0.248565	4.188076
DU - Reno - mild.	3964	26231	0.037666	0.347821	0.231594	3.900713
DU - NewReno - mild.	3967	26193	0.038401	0.353552	0.212080	4.265646
DU - Tahoe - mod.	9183	61475	0.036543	0.492887	0.499472	20.624833
DU - Reno - mod.	9107	61493	0.036564	0.494360	0.603107	20.641478
DU - NewReno - mod.	9133	61134	0.037043	0.488079	0.718285	20.182349
AL - Tahoe - uncon.	7725791	53127285	0.001019	0.001582	0.000000	0.000000
AL - Reno - uncon.	7646419	52911475	0.001018	0.001546	0.000000	0.000000
AL - NewReno - uncon.	7854495	53081298	0.001019	0.001550	0.000000	0.000000

Table 4.9: ns-3 results without initial congestion window functionality

Congestion level	Throughput (bps)		Avg. queuing delay		Loss rate (%)	
	L->R	R->L	L->R	R->L	L->R	R->L
DU - Tahoe - uncon.	2842	16089	0.0527	0.3650	0.8094	10.8239
DU - Reno - uncon.	2847	16210	0.0516	0.3659	0.7002	14.6317
DU - NewReno - uncon.	2866	16391	0.0476	0.3811	0.7892	15.3679
DU - Tahoe - mild.	4040	27192	0.0405	0.3758	0.5820	13.0230
DU - Reno - mild.	4063	27342	0.0394	0.3759	0.6115	16.8954
DU - NewReno - mild.	4087	27560	0.0380	0.3904	0.5661	17.8091
DU - Tahoe - mod.	9151	61548	0.0396	0.4903	0.9917	27.5638
DU - Reno - mod.	9246	61377	0.0399	0.4976	1.0929	30.3064
DU - NewReno - mod.	9230	60864	0.0400	0.4927	1.0044	29.9129
AL - Tahoe - uncon.	7814282	55793153	0.0010	0.0021	0.0000	0.0000
AL - Reno - uncon.	7677453	55071420	0.0010	0.0021	0.0000	0.0000
AL - NewReno - uncon.	7893047	55213328	0.0010	0.0021	0.0000	0.0000

Table 4.10: ns-3 original results

The TCP evaluation suite seemed like a good basis for evaluating the TCP functionality of each network simulator. The problem with the suite however was that it was a work in progress (and still is). The draft that we got from David Hayes was exactly that, a draft. This means that at several points, the suite is very unclear, which has made it difficult to work with at times.

Another big problem with choosing the TCP evaluation suite for our comparison of network simulators has been the traffic generation. Tmix is a very complex traffic generator, which makes it difficult to validate our results in general. How can we compare the results of ns-2, ns-3, and OMNeT++ if we cannot with absolute certainty claim that tmix works as intended? In talks with David we have learned that when they working on the evaluation suite in ns-2, they had a lot of trouble with tmix, and they had to make several improvements on it before they were able to use it in their suite. In this thesis we have also struggled a lot with tmix, both in ns-3, and in OMNeT++. We can conclude that when comparing simulators, tmix has been an unnecessarily complicating factor. To compare the simulators, a simpler traffic generator that would be easier to reproduce in all three simulators would have been a better choice.

4.7.1 Tmix

Through the work in this thesis, we have some thoughts on tmix as a traffic generator. First of all, tmix is a traffic generator that generates traffic based on connections described in a connection vector file. This means that the traffic not only depends on the generator itself, but also on the traces.

As mentioned above, we have scenarios where the average link utilization is 80%, but there still is not a single packet drop. This is *very* strange. One would think that even if the level of traffic is relatively low, TCP would still increase its sending rate in such a way that packet drops would occur. Because of this we would see a difference in the behaviour of Tahoe, Reno, and NewReno even though the level of traffic was low. However if there are no greedy flows, i.e., a flow that actually needs a high amount of bandwidth, and there are quite a few non-greedy flows, then it is in theory possible to have a high link utilization and no packet loss.

In real life, there is no such thing a real greedy flow, i.e., a flow that always tries to hog as much of the network resources as possible, but there are flows that are greedy in the more intuitive sense over a short amount of time. For instance the transfer of a 1GB file will act like a greedy flow for a while in a network with low capacity like 1Mbps. In this case, the TCP sender will try to hog as much of the capacity as possible for a long time. However, if we increase the network capacity to 10Gbps, then this transfer is not really greedy in the intuitive sense anymore, as there is more than enough capacity to transfer the file in a relative short amount of time, while still leaving network resources available for other flows.

This is part of what we think is the problem with tmix. Consider the creation of a connection vector file. To create the connection vector files, a tmix observer is placed somewhere in the network (preferably at

some bottleneck node of some sort). All the traffic that flows through this observer is saved in connection vector files, so that the traffic can be reproduced at a later point. Now consider what happens when a TCP sender wants to send a file larger than the maximum transmission unit (MTU), for instance 10MB. The TCP sender will split the file into several smaller segments. A typical MTU is 1500 bytes, with 40 bytes reserved for the TCP and IP headers. 10MB / 1500B give us 6850 packets. Now consider the network capacity to be 8Mbps. If we ignore the slow start and congestion avoidance algorithm for now, it should take 10 seconds to complete this transfer, where each packet arrives with an interval of roughly 1.5ms. If we try to replay this connection vector file in a similar network, then the traffic that tmix generates will look much like the traffic in the original network. It will look like a connection that is "greedy" for 10 seconds. Now what happens if we increase the capacity of the network? The problem is that the connection vector does not adapt to the new network. If we were to transfer 10MB in a higher capacity network, there would still be 6850 packets (given that the MTU is still the same), but the time between each of them would be much lower. Tmix does not reflect this.

The general problem here is that it is very difficult when observing a network to know whether a single packet is just a part of a large Application Data Unit (ADU), or whether it is an actual ADU in itself. In the previous example, the observer cannot know whether each of the packets observed are all part of a large ADU, or whether they are all individual smaller ADUs. When increasing the network capacity, there no longer exist any greedy flows in the connection vector files as the connection vector files are limited by the network where the trace was created. Traffic may be scaled by concentrating the arrival rate of connections, and as such it is possible to get congestion in the network, but it is also possible to get high network utilization, while still never dropping packets as there are no greedy flows.

Another problem with tmix is that it does not react to the state of the network. As the network congestion increases, TCP will back off to try to avoid a network collapse, but this is not the only thing that happens in real networks. For instance consider the users that surf the web or stream movies. What happens when packets are dropped repeatedly and the performance of the network stoops? A user might decide to watch the movie at a later time, or the user might give up on a web page that loads too slowly.

Even though tmix does not necessarily react exactly like the real world would as the network changes, it does not mean that tmix is useless as a tool to generate traffic. The traffic does have certain attributes from the real world, and the nature of the traffic is similar to traffic observed in the real world. We feel however that we are missing a few details, for instance the possibility of identifying greedy flows. It must be noted that it is easy to insert manually created connection vectors into the connection vector files that are greedy.

Chapter 5

Conclusion

The idea of this thesis was to try to investigate the TCP implementation of several commonly used network simulators to try to figure out whether they function like they are supposed to, i.e., as described in their respective RFCs, and also as seen in other research. There have been several problems however with this comparison, and the focus of this thesis has shifted more and more towards working with the TCP evaluation suite, rather than actually verifying the simulators.

We have implemented a subset of the tests described in the TCP evaluation suite in ns-3 and in OMNeT++. The reason for why we have not implemented the entire suite is partly because of the time constraint of this thesis, but also because we were getting a good indication of the results we could expect from only running the subset that we have implemented. We have compared these results to the ns-2 version of the suite. In addition to implementing the tests in each simulator, we have also created the tmix traffic generator in OMNeT++, as well as improved the tmix traffic generator in ns-3.

We have a number of results from ns-2, ns-3, and OMNeT++. In the results from ns-2 and ns-3 we can see the type of behaviour one would expect to see in the TCP variants that we have tested. The results from our OMNeT++ simulations do not show this kind of behaviour. If we compare the results from one simulator with the results of the others, we can see that the numbers deviate by quite a lot. Unfortunately, we cannot draw any conclusion to whether the simulators behave in the way they should for a number of reasons. The results stem from the same suite of tests, from the TCP evaluation suite. The TCP evaluation suite is very immature and is a work in progress. This fact has led to a bit of uncertainty when implementing the suite, which may have led to differences in the three simulator implementations. Another complicating factor has been the traffic generator tmix. We have worked a lot on making tmix work properly for ns-3, and we have also implemented this traffic generator from scratch in OMNeT++. Because our results deviate as much as they do, we believe that only element in the suite that may explain such a deviation is the traffic generator, which seems to have problems in its implementation. Because of this, we believe that the main cause of the differences we observe in the

three network simulators is caused by the traffic generator. As we suspect that tmix is not functioning identically in all three simulators, we cannot draw any conclusion to whether the simulators behave like the real world.

Even though we cannot verify the simulators, we have still worked a lot with the three network simulators, the TCP evaluation suite and tmix, and we do have some observations regarding these.

OMNeT++ is very limited in its TCP functionality. OMNeT++ has Tahoe, Reno, and NewReno, as well as TCP without congestion control built-in in the simulator. In addition to the built-in TCP algorithms, OMNeT++ also has added support for the Network Simulation Cradle (NSC) that makes it possible to run real world protocol stacks. The support for NSC is still very limited in OMNeT++. It only supports the Linux stack and the lightweight IP (lwIP) stack at the moment. We have not used lwIP, but we tried to use the Linux stack. NSC comes with a few limitations that make it difficult to use with tmix. For instance, it is not possible to change the maximum segment size (MSS), and it only supports the bytestream transfer mode, while tmix uses the object transfer mode.

ns-3 is also rather limited in its TCP functionality. ns-3, as OMNeT++, has built-in support for Tahoe, Reno and NewReno, as well as TCP without congestion control. There is no support for selective acknowledgements (SACK) or Nagle's algorithm. These three very basic TCP algorithms without support for SACK make for a very limited TCP functionality in ns-3. In addition to the TCP algorithms however, ns-3 also supports NSC. NSC in ns-3 has its limitations as well. First of all it only supports Linux stacks, and it has some other limitations as well, such as the fact that NSC only works on single-interface nodes. We were unable to run NSC in this thesis because NSC crashed in almost every single simulation except for the really short-running ones (the dial-up link scenario). NSC in ns-3 is still a work in progress however, and we will probably see improvements in the future.

ns-2 is very rich in its TCP functionality compared to ns-3 and OMNeT++. ns-2 has built-in support for the more basic TCP variants such as Tahoe, Reno and NewReno, but it also has built-in all the variants found in the Linux kernel, which includes for instance Westwood, BIC, CUBIC and Hybla. In addition to the built-in variants, it is also possible to run the NSC in ns-2 (actually, NSC was built for ns-2), and has no limitations like the ones seen in ns-3 and OMNeT++ as far as we know.

We have, as far as we know, been the first to try the suite in practice, and in our opinion it is not ready yet. While the idea of creating a standardized test suite to evaluate TCP is a good one, there is still room for improvement in the current state of the suite.

One problem with the suite is the way that it defines congestion. Congestion is simply defined as an average drop rate. For instance, a scenario is defined to be moderately congested when the average loss rate of the simulation meets a certain value. The problem with defining congestion in this way is that the TCP streams are constantly on the edge of collapsing, which makes them very unstable. In short simulation runs it is possible to find a level of traffic so that the average loss rate meets the target

loss rate. In long simulation runs however, it is very difficult to maintain a level of congestion without the network collapsing at some point. Another problem with this way of defining congestion is that each TCP variant behaves differently, making it impossible for every TCP variant to meet the target loss rate at the same time. The evaluation suite is very vague at this point.

Another concern we have for the TCP evaluation suite is the way that traffic is generated. At the moment the tmix traffic generator is used. Tmix is based on network traces from real networks, and is thus realistic in that sense. We argue however, that when the network changes (for instance if we increase the capacity of the network), the traffic generated by tmix does not reflect this. We believe that the traffic generated by tmix is limited by the network in which the traces were captured.

5.1 Future work

Both ns-3 and OMNeT++ are very limited when it comes to TCP. We believe that once NSC is properly in place, this will improve. Then there is also the question of whether the simulators act the way that they should. We have found several unexpected results in our simulations that could be further explored, for instance, why does Reno in some cases outperform NewReno?

We have also worked quite a bit with the tmix traffic generator. One of the problems with generating traffic from trace sources is that it is very difficult when observing a network to reverse engineer what the application was originally doing. For instance, when a client requests a large file, the observer will see a steady stream of smaller segmented packets coming from the server, but there is no way for the observer to know what the original request looked like. This is seen clearly in tmix. Tmix will reproduce the traffic as seen in the network, so if the interarrival time of each packet observed was 1sec, then tmix will reproduce this. When the capacity of the network increases however, the interarrival time of each packet should be decreased, but tmix does not reflect this. It would have been useful to have a traffic generator that could reverse engineer the traffic observed in a better way than tmix does, although we acknowledge the fact that this would be very difficult to do. Another perhaps simpler solution would be to figure out a better way to scale tmix traffic, so that is reflects changes in the network configuration in a better way than it does at the moment.

The TCP evaluation suite requires some work in its current state. Many of the problems stem from the way that the suite defines congestion. A fundamentally different way of handling congestion is required before the suite can be used as a reliable tool. No one argues the value of having a standardized test suite to evaluate TCP extensions. We believe that this is an important contribution to the research community. The TCP evaluation suite in its current state however, has some way to go.

Appendix A

ns-3 README

A.1 Prerequisites and installation

The version of ns-3 used in this thesis, is version 3.14.1. A basic knowledge of ns-3 is assumed (installing ns-3 and at least being able to run examples). In addition to the simulator itself, we use an additional model, the tmix model. We have extended the functionality of tmix in this thesis, and our improved version is included with the suite. The folders *tmix* and *delaybox* are to be placed in the src folder of ns-3. The folder *eval*, which includes the simulation suite itself, is to be placed in the examples folder of ns-3. Because the suite is implemented as an example, examples must be enabled in ns-3 for the suite to be compiled. This can be enabled by doing *'./waf configure -d optimized –enable-examples'*.

The evaluation suite expects to find an environment variable named *NS3EVAL* that contains the path to the base folder of the suite, and it also expects to find an environment variable named *NS3* that contains the path to the base folder of ns-3 (usually something like /home/user/matshr/ns-allinone-3.14.1/ns-3.14.1/).

The connection vector files are the same as for the ns-2 version of the suite, and can be found at http://caia.swin.edu.au/ngen/tcptestsuite/tools.html. In the file dumbbell.cc, the path to these files must be given to each tmix application (just change the paths that are currently in the file).

A.2 Usage

There is a convenient script included with the ns-3 evaluation suite: tcp-eval.sh. Typing './tcp-eval.sh help' gives an overview of the commands available. Starting a scenario is just a matter of typing './tcp-eval.sh <testName>', for instance './tcp-eval-sh accessLink'. The results are automatically placed in the results folder in CSV format.

To change the TCP version to run tests with, there is a parameter named tcpVersion that the simulation script can be started with. For instance './waf –run 'basic-scenario –testName=accessLink –tcpVersion=TcpReno'. The main script does not include this functionality, so it is probably just

as easy to open the file named *basic-scenario.cc* and change the name of the variable testName to whatever TCP version to test.

Appendix B

OMNeT++ README

B.1 Prerequisites

There are not many prerequisites for using the OMNeT++ evaluation suite. The version used in this thesis is made for OMNeT++ version 4.2.2, and uses the INET framework version 2.0 in addition to the basic simulator. In addition to the simulator itself, there is a package named tmix included with the evaluation suite that needs to be installed. The folder *tmix* is to be placed in the *src* folder of INET. Tmix expects to find an environment variable named *VECTORFILES*, which should contains the path to a directory where each of the connection vector files are found. The connection vector files are the same as for the ns-2 version of the suite, and can be found at http://caia.swin.edu.au/ngen/tcptestsuite/tools.html.

B.2 How to run the evaluation suite in OMNeT++

It is easiest to compile and run the suite from the IDE that comes with OMNeT++. To do this, place both the eval project and the inet project in the same directory. Import both of these in the IDE as existing projects. To eval project must reference the inet project. This is done under project references found in the project properties of eval. To run a scenario, right-click the omnepp.ini file of eval and select run as -> OMNeT++ simulation.

Analysing the results is not very intuitive. We have used the analysis tool built into the IDE. The problem with this is that it is not possible to use this analysis tool from the command line, thus making it impossible to make a script that first runs the simulation, then analyses the results. The file dumbbell.anf is the analysis file we have used. To use this file, open it, select the input tab, drag a results file (.sca) into the input files window, select the datasets tab, right click the data set and select export data as CSV. Doing this will select a few selected attributes from the results and print these in CSV format. Choose a name for this output file. There is a script named *analysisScript.sh* that can be used to calculate the final results of the simulation. The script is used like this: './analysisScript.sh <testName> <testFile>'. For instance './analysisScript.sh accessLink AL_uncon'. This is given that the CSV file from the analysis tool is named AL_uncon.

The script will look in the default output folder of OMNeT++ (which is .../omnet-4.2.2/ide)

Using the tool is a bit unintuitive as it was not ment to be released. If anything is unclear, do not hesitate to mail me at matshr@ifi.uio.no.

Appendix C

Contents of CD-ROM

Included on the CD-ROM are our TCP evaluation suite implementations for ns-3 and OMNeT++, as well as tmix for both of them. Instructions on how to use the suite is included in the two previous appendices, as well as in the README's included with the suites.

Bibliography

[1] Acm digital library. http://dl.acm.org/.

[2] Bic and cubic. http://research.csc.ncsu.edu/netsrv/?q=content/bic-and-cubic. [Online; accessed 05.10-2012].

[3] Estinet. http://www.estinet.com/.

[4] Google. http://www.google.com.

[5] Inet framework. http://inet.omnetpp.org/.

[6] Libpcap file format. http://wiki.wireshark.org/Development/LibpcapFileFormat. [Online; accessed 23.05-2012].

[7] A lightweight tcp/ip stack. http://savannah.nongnu.org/projects/lwip/. [Online; accessed 11.11-2012].

[8] Network simulation cradle. http://research.wand.net.nz/software/nsc.php. [Online; accessed 03.11-2012].

[9] The ns-2 tcp evaluation suite. http://caia.swin.edu.au/ngen/tcptestsuite/tools.html.

[10] Omnet++ user manual. http://www.omnetpp.org/doc/omnetpp/manual/usman.html. [Online; accessed 29.05-2012].

[11] Opnet. http://www.opnet.com/.

[12] R. http://www.r-project.org/. [Online; accessed 22.10-2012].

[13] Tmix-ns3. http://code.google.com/p/tmix-ns3/.

[14] Valgrind. http://valgrind.org. [Online; accessed 03.11-2012].

[15] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC 2581 (Proposed Standard). Obsoleted by RFC 5681, updated by RFC 3390.

[16] Doreid Ammar, Thomas Begin, and Isabelle Guerin-Lassous. A new tool for generating realistic internet traffic in ns-3. http://perso.ens-lyon.fr/thomas.begin/Publis/SIMUTools11.pdf. [Online; accessed 01.11-2012].

[17] Lachlan Andrew, Cesar Marcondes, Sally Floyd, Lawrence Dunn, Romaric Guillier, Wang Gang, Lars Eggert, Sangtae Ha, and Injong Rhee. Towards a common tcp evaluation suite. In *PFLDnet 2008*, March 2008.

[18] Martin Bateman, Saleem Bhatti, Greg Bigwood, Devan Rehunathan, Colin Allison, Tristan Henderson, and Dimitrios Miras. A comparison of tcp behaviour at high speeds using ns-2 and linux. In *Proceedings of the 11th communications and networking simulation symposium*, CNS '08, pages 30–37, New York, NY, USA, 2008. ACM.

[19] Kevin Fall and Sally Floyd. Simulation-based comparisons of tahoe, reno and sack tcp. *SIGCOMM Comput. Commun. Rev.*, 26(3):5–21, July 1996.

[20] S. Floyd, T. Henderson, and A. Gurtov. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 3782 (Proposed Standard).

[21] Sally Floyd and Vern Paxson. Difficulties in simulating the internet. *IEEE/ACM Trans. Netw.*, 9:392–403, August 2001.

[22] David Hayes, Sally Floyd, and Wang Gang. Common tcp evaluation suite. http://tools.ietf.org/html/draft-irtf-tmrg-tests-02. [Online; accessed 22.10-2012].

[23] Teerawat Issariyakul and Ekram Hossain. *Introduction to Network Simulator NS2*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[24] Sam Jansen and Anthony McGregor. Validation of simulated real world tcp stacks. https://docs.google.com/present/view?id=dfk8s4n7_47ff5hgk6d&pli=1. [Online; accessed 18.11-2012].

[25] Woratat Makasiranondh, S. Paul Maj, and David Veal. Pedagogical evaluation of simulation tools usage in network technology education. *Engineering and Technology*, 8:321–326, 2010.

[26] John Nagle. Congestion control in ip/tcp internetworks. *SIGCOMM Comput. Commun. Rev.*, 14(4):11–17, October 1984.

[27] The network simulator ns-2. http://www.isi.edu/nsnam/ns/doc/index.html. [Online; accessed 25.09-2012].

[28] Oracle. Oracle vm virtualbox. https://www.virtualbox.org/, 2010. [Online; accessed 24.05-2012].

[29] Vern Paxson. End-to-end internet packet dynamics. *SIGCOMM Comput. Commun. Rev.*, 27(4):139–152, October 1997.

[30] J. Postel. RFC 768: User Datagram Protocol, August 1980.

[31] J. Postel. RFC 793: Transmission Control Protocol, September 1981.

[32] Zhao A. Qun and Wang Jun. Application of ns2 in education of computer networks. In *Proceedings of the 2008 International Conference on Advanced Computer Theory and Engineering*, pages 368–372, Washington, DC, USA, 2008. IEEE Computer Society.

[33] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream control transmission protocol, 2000.

[34] Andrew Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 4th edition, 2002.

[35] Klaus Wehrle, Mesut Günes, and James Gross, editors. *Modeling and Tools for Network Simulation*. Springer, 2010.

[36] Michele C. Weigle, Prashanth Adurthi, Félix Hernández-Campos, Kevin Jeffay, and F. Donelson Smith. Tmix: a tool for generating realistic tcp application workloads in ns-2. *SIGCOMM Comput. Commun. Rev.*, 36(3):65–76, July 2006.

[37] Wikipedia. Compatibility mode (software). http://en.wikipedia.org/wiki/Compatibility_mode_%28software%29, February 2012. [Online; accessed 24.05-2012].